Guide to the File System Manager ®

Version 1.2

January 6, 1995 (revision 2)

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

1 THE FILE SYSTEM MANAGER

ABOUT THIS CHAPTER

This chapter describes foreign file systems and the File System Manager (FSM). It also provides a complete description of all FSM data types, FSM service routines available to a foreign file system and other programs, and the FSM component interfaces.

ABOUT FOREIGN FILE SYSTEMS

A foreign file system allows Macintosh applications to gain access to non-Macintosh volumes using File Manager routines. For example, a Macintosh application can use File Manager routines to read from and write to files on a ProDOS disk if there is a foreign file system for ProDOS.

In the past, developing a foreign file system required extensive knowledge of the Macintosh File Manager and how it used both documented and undocumented low-memory global variables and data structures. To solve this problem, Apple has written the File System Manager. To create a new foreign file system, developers no longer need to access undocumented portions of the Macintosh and interface with the Macintosh file system through a 68000 register-based interface. Instead, they provide a foreign file system for a particular file system that works with the File System Manager. The File System Manager provides a systematic way for one or more foreign file systems to interact with the Macintosh file system using high-level language interface.

The File System Manager performs low-level tasks common to all foreign file systems. It intercepts incoming file system related traps, identifies the foreign file system the request should be passed to, and then passes on only those requests requiring action by the particular foreign file system. To develop a foreign file system, you provide a set of foreign file system routines—described in the following chapters—that the File System Manager can call.

> **Important Note:** Even though the File System Manager provides many services that simplify development of foreign file systems, developing a foreign file system is both a difficult and time-consuming process. A minimal foreign file system must implement over forty Macintosh file system routines while a networked, sharable file system will have to implement as many as eighty Macintosh file system routines. To write a foreign file system you must be familiar with the low-level Macintosh file system data routines and data structures described in *Inside Macintosh: Files*and with the material covered in the chapters of the *Guide to the File System Manager.*

ABOUT THE FILE SYSTEM MANAGER

The File System Manager is the part of the Macintosh Operating System that manages the use of foreign file systems. The File System Manager provides a general means by which foreign file systems can be installed, identified, and interfaced to the Operating System .

The Operating System services provided by the File System Manager are called FSM components and the interface mechanism between an foreign file system and a particular FSM component is known as a FSM component interface. At this time, two FSM component interface are defined for use under the File System Manager

- the HFS Component Interface which allows a foreign file system to process File Manager requests
- the Disk Initialization Package Component Interface which allows foreign file systems to initialize foreign file system volumes on a Macintosh

For each installed file system, the File System Manager maintains a file system descriptor (FSD). Information in the FSD identifies the file system and describes its interfaces to the File System Manager and to the FSM component interfaces supplied by the File System Manager. Installed FSDs are kept in the File System Manager's private FSD queue.

A foreign file system's FSD can be installed, accessed, maintained, and removed with the FSM service routines. FSM service routines also allow applications to communicate with the File System Manager and foreign file systems, and allow disk drivers and foreign file systems to communicate with the File System Manager

Figure 1-1 shows how the Operating System and the File System Manager work together to allow applications to access foreign file systems.

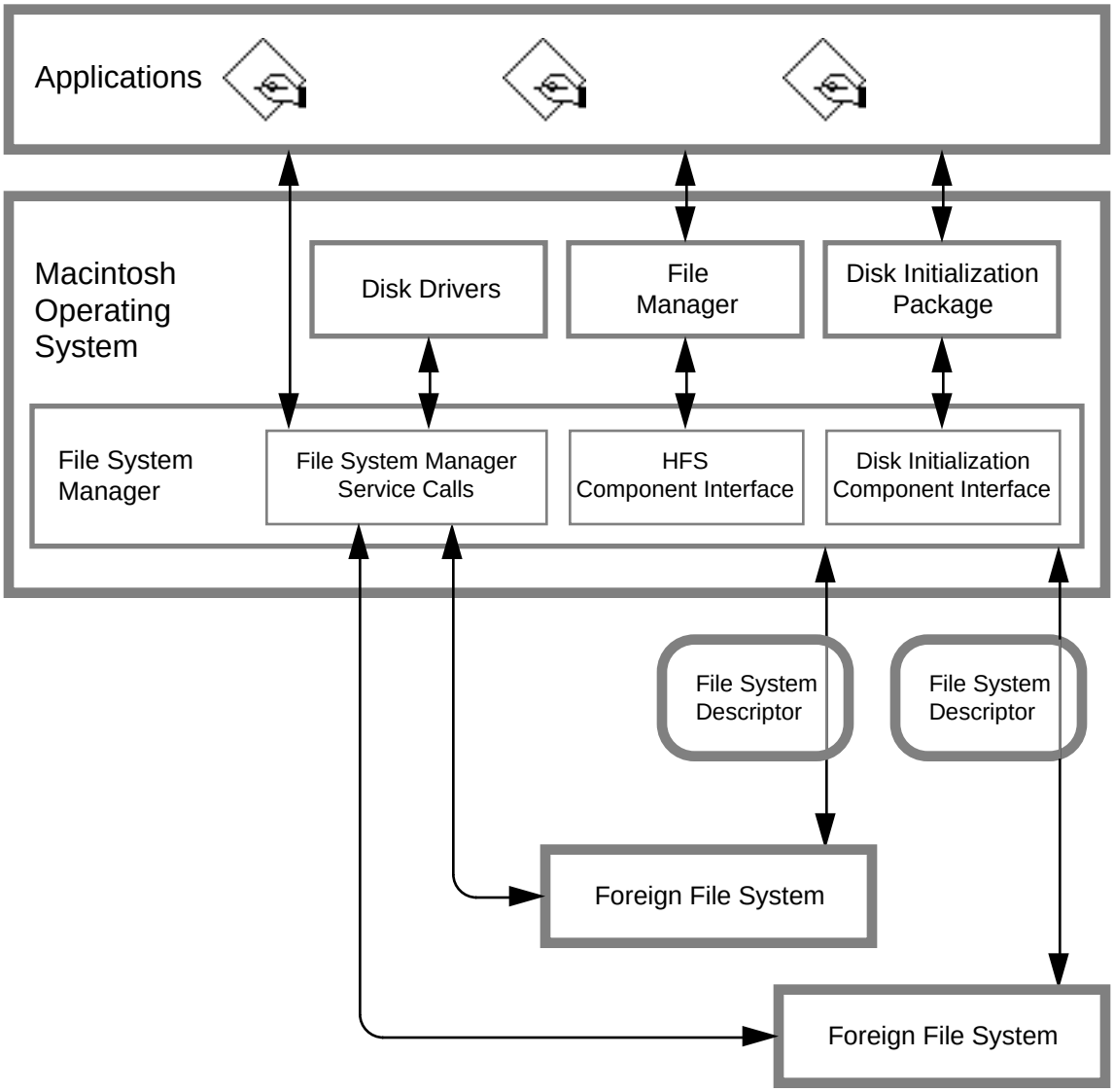

Figure 1-1. Applications using foreign file systems through the File System Manager

USING THE FILE SYSTEM MANAGER

This section describes

- File system descriptors (FSDs)
- FSM component interfaces
- how to determine if the File System Manager is available
- how foreign file systems are installed and enabled

File System Descriptors

The File System Manager maintains a file system descriptor (FSD) for each installed foreign file system. Information in the FSD identifies the file system and describes its interfaces to the FSM component interfaces supplied by the File System Manager.

A foreign file system's FSD is installed, accessed, maintained, and removed with the FSM service routines. FSDs are created with InstallFS. A copy of the information in an FSD can be obtained with GetFSInfo. Information in an FSD can be modified with SetFSInfo. An FSD can be removed with RemoveFS. The FSM service routines use the data type FSDRec.

```
struct FSDRec {
     struct FSDRec   *fsdLink;             /* ptr to next */
     short           fsdLength;            /* length of this FSD in
                                              bytes */
     short           fsdVersion;           /* version number */
     short           fileSystemFSID;       /* file system id */
     Str31           fileSystemName;       /* file system name */
     FSSpec          fileSystemSpec;       /* foreign file system's
                                              FSSpec */
     Ptr             fileSystemGlobalsPtr; /* ptr to file system
                                              globals */
     FSDCommUPP      fileSystemCommProc;   /* communication proc with
                                              the FFS */
     long            reserved3;            /* reserved, must be 0 */
     long            reserved2;            /* reserved, must be 0 */
     long            reserved1;            /* reserved, must be 0 */
     HFSCIRec        fsdHFSCI;             /* HFS component
                                              interface */
     DICIRec         fsdDICI;              /* Disk Initialization
                                              component interface */
     };
     typedef struct FSDRec FSDRec;
     typedef FSDRec *FSDRecPtr;
```

Field descriptions

fsdLink	A pointer to the next entry in the FSD list.
fsdLength	The length of of this FSD in bytes.
fsdVersion	The lowest version of the File System Manager this foreign file system can support. Currently, this is 1.
fileSystemFSID	The unique file system ID (FSID) of the foreign file system. See “File System ID Numbers” on page 1-7 for more information about FSID numbers.
fileSystemName	A Pascal string (Str31) which is used to identify the foreign file system. It is not guaranteed to be unique. fileSystemName is used by the Disk Initialization Package. The name is displayed in a dialog box that presents the user a choice of file system formats with which a disk may be initialized.
fileSystemSpec	An FSSpec indicating the location of the foreign file system’s file code file. The File System Manager fills in fileSystemSpec with the location of the current resource file (CurResFile) when the FSD is installed by InstallFS.
fileSystemGlobalsPtr	A pointer to the foreign file system’s optional global data area. It is passed to the foreign file system as a parameter for all FSM component interface requests to the foreign file system. A nil value indicates no global area was allocated. The global data area is allocated by the foreign file system installation code.
fileSystemCommProc	A pointer to the foreign file system’s communications function.
reserved3	Reserved. Must be zero.
reserved2	Reserved. Must be zero.
reserved1	Reserved. Must be zero.
fsdHFSCI	The foreign file system’s HFS component interface record. See “The HFS Component Interface Record” in Chapter 2 for more information about the HFS component interface record.
fsdDICI	The foreign file system’s Disk Initialization component interface record. See “The Disk Initialization Component Interface Record” in Chapter 4 for more information about the Disk Initialization component interface record.

File System ID Numbers

A file system ID (FSID) number is a unique word-length value that identifies a particular file system. This number is used by the Macintosh file system and by the File System Manager to identify a particular file system. The FSID value passed to InstallFS is checked by the File System Manager to insure that a FSD with that FSID is not already installed. The FSID is passed as a parameter to all File System Manager service routines. Table 1-1 is a list of file system IDs currently owned by Apple file systems.

Table 1-1. File System IDs owned by Apple file systems

FSID	File System
$0000	Macintosh HFS or MFS
$0100	ProDOS File System
$0101	PowerTalk Mail Enclosures
$4147	ISO 9660 File Access (through Foreign File Access)
$4242	High Sierra File Access (through Foreign File Access)
$464D	QuickTake File System (through Foreign File Access)
$4953	Macintosh PC Exchange (MS-DOS)
$4A48	Audio CD Access (through Foreign File Access)
$4D4B	Apple Photo Access (through Foreign File Access)

FSID values between $0001 and $00FF are reserved for old-style foreign file systems that determine their FSID's dynamically (i.e., by searching the drive queue) and cannot be used for File System Manager foreign file systems.

FSID values between $0100 and $7FFF are reserved for the File System Manager and other foreign file systems that use registered file system ID numbers.

FSID values $8000 and above are reserved for future use by Apple Computer, Inc.

FSID values fsmIgnoreFSID ($FFFE) and fsmGenericFSID ($FFFF) have special meaning. fsmIgnoreFSID is stored in the partition map on a partitioned disk. It signals the disk driver not to install this partition into the drive queue. fsmGenericFSID is stored in the drive queue element (DrvQEl) by the disk driver to signal an unknown foreign file system volume. The File System Manager will attempt to find a foreign file system that can identify disks with the fsmGenericFSID by calling each foreign file system with the ffsIDVolMountMessage message. This allows disk drivers to support multiple volume formats without having to identify the volume format.

To reserve a FSID in the File System Manager range ($0100 through $7FFF), a special creator type must be registered with Apple's Developer Support Center (AppleLink: DEVSUPPORT). The high word of the creator type must be $0613. The low word of the creator type is the FSID to register. For example, to reserve the FSID $0100 for a foreign file system, the creator type $06130100 should be registered. The creator type registered for an FSID can also be used as the creator type of the system extension file that implements a foreign file system.

Important: File System ID numbers in the File System Manager range were not registered before 1993. As a result, there are foreign file systems in use that have not registered with Apple's Developer Support Center. If you are the publisher of one of

those foreign file systems, please register your FSID immediately to help prevent file system ID conflicts.

File System Communications Function

The communications function pointed to by fileSystemCommProc in an FSDRec is used by the File System Manager to communicate with a foreign file system. When the file system communications function is called, it is passed a message and a pointer to a buffer containing any additional parameters needed to process the message. The currently defined messages are listed in Table 1-2.

Table 1-2. File System Communications Function Messages

Message		Purpose
ffsNopMessage	0	No operation. The file system communications function should simply return a result of noErr.
ffsGetIconMessage	1	Return disk icon and mask.
ffsIDDiskMessage	2	Identify the volume about to be mounted.
ffsLoadMessage	3	Load the foreign file system's HFS component interface code and data.
ffsUnloadMessage	4	Unload the foreign file system's HFS component interface code and data.
ffsIDVolMountMessage	5	Identify a VolMountInfo record by its media field.
ffsInformMessage	6	Foreign file system defined message.

The file system communications function is described in detail later in this chapter.

Global Data Area

Unlike a Macintosh application, a foreign file system possesses no A5 world. Therefore, a foreign file system cannot define and use global variables. Instead, the File System Manager allows a foreign file system to allocate a global data area which is used to store data needed globally by the foreign file system.

The global data area must be allocated in the system heap before the foreign file system is installed and the address of the global data area is stored in the foreign file system's FSD. Every time the File System Manager calls a foreign file system function, the address of the global data area is passed to the foreign file system as one of the function parameters. This allows any part of a foreign file system to access the data in its global data area.

FSM Component Interfaces

The File System Manager supplies an interface mechanism known as an FSM component interface. An FSM component interface can be any functional interface exported by a given Macintosh operating system or toolbox component. Each FSM component interface is an independent interface, defined and managed by a particular Macintosh operating system or toolbox component. This includes the definition of the calls involved, the dispatching of calls to the foreign file systems, and the execution environment in which those calls operate.

At this time, two FSM component interfaces are defined for use under the File System Manager

- the HFS Component Interface which allows a foreign file system to process File Manager requests
- the Disk Initialization Package Component Interface which allows foreign file systems to initialize foreign file system volumes on a Macintosh

The FSD and the FSM component interface mechanism are both designed so that new FSM component interfaces can be added by Apple Computer to the File System Manager at a later time if needed.

The connection between a given component and a foreign file system is defined by a FSM component interface record which is contained in a foreign file system's FSD. A minimum FSM component interface record includes a dispatch mask which controls the use of the interface, and at least one pointer to the foreign file system code responsible for processing the requests from that component. Additional interface parameters other than the dispatch mask and code pointer may be included in a given FSM component interface. These are FSM component interface dependent parameters and are not required by the File System Manager. These parameters are later used by the individual operating system or toolbox components to dispatch their requests to the foreign file system.

FSM Component Interface Dispatch Mask

The first long word of an FSM component interface record is the FSM component interface dispatch mask (compInterfMask). The bits of the compInterfMask currently have these meanings:

Bit	Name	Meaning
0-23		FSM component interface dependent flags. Each component is free to define these flag bits as needed.
24-29		Reserved for the File System Manager's use.
30	fsmComponentBusyBit	If set, the FSM component interface is busy (i.e., one or more requests are outstanding). This bit is maintained by the component and used by the File System Manager to control the use of the SetFSInfo File System Manager service routine.

31	fsmComponentEnableBit	If set, the component may begin dispatching requests to the foreign file system. The foreign file system should set this bit with the SetFSInfo File System Manager service routine once it is ready to receive requests.

FSM Component Interface Processing Function

The second long word of an FSM component interface record is a pointer (compInterfProc) to the foreign file system code responsible for processing redirected operating system or toolbox requests. Which operating system or toolbox requests are redirected is part of the FSM component interface definition.

FSM Component Interface Dispatching Conventions

In order to insure some degree of uniformity across FSM component interfaces, the File System Manager imposes some common conventions for use by components when dispatching requests to a foreign file system. Those conventions are

- The global pointer (fileSystemGlobalsPtr) from the FSD record is passed to the foreign file system on all requests. The global pointer is needed to locate context information associated with the operation of the foreign file system.
- The FSM component interface should dispatch requests to a foreign file system only if the fsmComponentEnableBit flag in the FSM component interface dispatch mask is set.
- The FSM component interface will set the fsmComponentBusyBit flag in the target file system's FSM component interface dispatch mask when the interface is dispatched and clear the flag when the request has completed. This will prohibit modification of FSM component interface parameters by SetFSInfo.

Ensuring the File System Manager is Available

You must ensure the File System Manager is available on the user's computer before calling the File System Manager. You can determine is the File System Manager is available by using the Gestalt function. The Gestalt selector is gestaltFSAttr. The File System Manager is available if the Gestalt function returns a result of noErr and the gestaltHasFileSystemManager bit is set in the response parameter.

In addition, you must ensure the File System Manager is version 1.2. Earlier versions of the File System Manager do not implement all of the interfaces described in this manual.

Listing 1-1 illustrates how you use Gestalt to determine if the File System Manager is available and ensure that the File System Manager is version 1.2 or later.

Listing 1-1. Testing for the required version of the File System Manager

```
static   Boolean  HasRequiredFSM(void)
{
   long  response;
   Boolean  result;

   result = false;
   /* Make sure the File System Manager is installed */
   if ( Gestalt(gestaltFSAttr, &response) == noErr )
   {
      if ( (response & (1L << gestaltHasFileSystemManager)) != 0 )
      {
         /* We require File System Manager 1.1 features so */
         /* check the version of FSM. */
         if ( Gestalt(gestaltFSMVersion, &response) == noErr )
         {
            /* Make sure we have File System Manager 1.2 or later */
            if ( (unsigned long)response >= 0x0120)
               result = true;
         }
      }
   }
   return ( result );
}
```

Installing and Enabling a Foreign File System

Installation of foreign file systems will usually be accomplished by a system extension (INIT) file's code at startup time. The mechanism that executes a system extension (commonly called the INIT 31 mechanism) is described in Chapter 29 "The System Resource File" in *Inside Macintosh Volume IV*, in Chapter 19 "The Start Manager" in *Inside Macintosh Volume V*, and in *Inside Macintosh: Operating System Utilities* The INIT 31 mechanism will load and execute the foreign file system installation code in an 'INIT' resource in the foreign file system file.

The installation can also be done from within an application if precautions are taken to ensure the foreign file system is installed in the system heap, or if installed in the application's heap, that all volume are unmounted and the foreign file system removed before the application quits.

If the File System Manager is available (see the section, Ensuring the File System Manager is Available) then the foreign file system installation code should install the foreign file system. To install a foreign file system, the foreign file system initialization code must allocate space for the global data area, load and detach the code resource for the FSM component interface processing function, initialize the fields of an FSDRec, and call InstallFS to add an FSD to the File System Manager's FSD queue. If InstallFS is successful, the foreign file system is installed with its component interfaces disabled. Once the foreign file system is installed, the next step is to initialize the component interface records in the FSD to let the File System Manager know the features the foreign file system is capable of.

All foreign file systems must support the HFS component interface. To initialize the HFSCIRec in the FSD, the installation code should set the hfsCIDoesDynamicLoadMask bit in the compInterfMask field of the HFSCIRec. The File System Manager will call the fsdCommProc function with a ffsLoadMessage when it needs the foreign file system's HFS component interface code loaded.

If the foreign file system supports the Disk Initialization component interface, then the installation code needs to load and detach the foreign file system's disk initialization code and data, and then enable the Disk Initialization component interface in the FSD.

If the foreign file system is successfully installed, the installation code should call InformFSM with a fsmDrvQElChangedMessage message to mount any volumes owned by the foreign file system that might already be in drives.

FILE SYSTEM MANAGER SERVICE ROUTINES

Six FSM service routines are provided by the File System Manager. These service routines let you install, access, maintain and remove the information in the FSDs, send the File System Manager messages, and send file system specific messages to a foreign file system. The FSM service routines are listed in Table 1-3.

Table 1-3. File System Manager Service Routines

Name	Purpose
InstallFS	Adds a new FSD to the File System Manager's FSD queue.
RemoveFS	Removes a FSD from the File System Manager's FSD queue.
GetFSInfo	Returns the FSD for a specific foreign file system or all foreign file systems.
SetFSInfo	Changes the FSD information of a foreign file system.
InformFSM	Sends messages or requests to the File System Manager.
InformFFS	Sends a file system specific message to a foreign file system.

All of the FSM service routines execute synchronously. GetFSInfo and SetFSInfo may be called at any time. InstallFS, RemoveFS, and InformFSM cannot be called by code executing at interrupt time — they may make Memory Manager requests that move memory, or synchronous operating system requests. Note that the File System Manager service routines are not Macintosh file system routines and are not controlled by the file system queuing mechanism.

InstallFS

Use the InstallFS function to add a new file system descriptor to the system.

```
pascal short InstallFS (FSDRecPtr fsdPtr);
```

fsdPtr — Contains a pointer to an FSDRec initialized with the data you want copied to the file system descriptor InstallFS creates in the FSD queue.

fsdPtr record

→ fsdLength	short	Contains the size of the FSDRec. Should always be sizeof(FSDRec).
→ fsdVersion	short	Contains the minimum version of the File System Manager this foreign file system will work with. For the first release, use fsdVersion1.
→ fileSystemFSID	short	Contains the unique file system ID (FSID) of the foreign file system.
→ fileSystemName	Str31	Conatins a Pascal string which is used to identify the foreign file system by name.
→ fileSystemGlobalsPtr	Ptr	Contains a pointer to the foreign file system's optional global data area.
→ fileSystemCommProc	FSDCommUPP	Contains a pointer to the foreign file system's communications function.
→ reserved3	long	Reserved field. Must be 0.
→ reserved2	long	Reserved field. Must be 0.
→ reserved1	long	Reserved field. Must be 0.
→ fsdHFSCI	HFSCIRec	Contains the foreign file system's HFS component interface record.
→ fsdDICI	DICIRec	Contains the foreign file system's Disk Initialization component interface record.

The InstallFS function

- validates the FSDRec pointed to by fsdptr
- allocates a new file system descriptor
- copies the FSDRec pointed to by fsdptr to the new file system descriptor
- clears the fsdHFSCI and fsdDICI compInterfMask fields in the new file system descriptor
- initializes fileSystemSpec in the new file system descriptor to the location of the resource file referenced by CurResFile
- adds the new file system descriptor to the File System Manager's FSD queue

Note that the FSDRec passed by fsdptr is only copied by InstallFS. The FSDRec passed by fsdptr does not become part of the File System Manager's FSD queue.

Result codes

noErr	0	No error
fsmBadFFSNameErr	-433	length of fsdPtr->fileSystemName is 0 or greater than 31
fsmBadFSDLenErr	-434	The file system descriptor is too large (fsdPtr->fsdLength is greater than sizeof(FSDRec))
fsmDuplicateFSIDErr	-435	An FSD with fsdPtr->fileSystemFSID already exists in FSD queue
fsmBadFSDVersionErr	-436	fsdPtr->fsdVersion is greater than the File System Manager version
memFullErr	-108	The new file system descriptor could not be allocated

RemoveFS

Use the RemoveFS function to remove file system descriptor from the File System Manager's FSD queue.

```
pascal short RemoveFS (short fsid);
```

fsid Contains the unique file system ID (FSID) of the foreign file system.

RemoveFS removes the FSD specified by fsid from the File System Manager's FSD queue and deallocates the memory allocated by the File System Manager for the file system descriptor. The file system descriptor can be removed only if all volumes owned by the foreign file system are unmounted and all FSM component interfaces in the file system descriptor are not busy.

Result codes

noErr	0	No error
fsmBusyFFSErr	-432	A volume with the fsid is mounted or an FSM component interfaces is busy
fsmFFSNotFoundErr	-431	The file system descriptor with fsid was not found in the FSD queue

GetFSInfo

Use the GetFSInfo function to get a copy of a file system descriptor in the File System Manager's FSD queue.

```
pascal short GetFSInfo (short selector, short key, short *bufSize,
                        FSDRecPtr fsdptr);
```

selector	Contains the method used to select the file system descriptor.
key	Contains the key used to select the file system descriptor.
bufSize	Contains a pointer to a short. On input, the field referred to by this parameter contains the size of the buffer pointed to by fsdPtr; on output, GetFSInfo places the number of bytes actually copied from the file system descriptor to the buffer into the field referred to by this parameter.
fsdPtr	Contains pointer to a FSDRec. GetFSInfo copies the file system descriptor into the FSDRec referred to by this parameter.

The GetFSInfo function returns a copy of an file system descriptor in the FSD queue. The GetFSInfo function selects the file system descriptor according to these rules:

- If selector is fsmGetFSInfoByIndex, GetFSInfo returns returns a copy of the file system descriptor whose position in the FSD queue is specified by the key parameter. A key of 0 returns a copy of the first file system descriptor in the FSD queue.
- If selector is fsmGetFSInfoByFSID, GetFSInfo returns returns a copy of the file system descriptor whose File System ID is specified by the key parameter.
- If selector is fsmGetFSInfoByRefNum, GetFSInfo returns returns a copy of the file system descriptor using the volume or file reference number specified by the key parameter. The key parameter must be a valid file or volume reference number and must be a file or volume owned by a foreign file system installed by the File System Manager.

Result codes

noErr	0	No error
rfNumErr	-51	The key was an invalid reference number or referenced a volume not controlled by the File System Manager
fsmFFSNotFoundErr	-431	The file system descriptor with the FSID, or at the specified index was not found in the FSD queue

SetFSInfo

Use the SetFSInfo function to change a file system descriptor in the File System Manager's FSD queue.

```
pascal short SetFSInfo (short fsid, short bufSize, FSDRecPtr fsdptr);
```

fsid	Contains the unique file system ID (FSID) of the foreign file system.
bufSize	Contains the size of the buffer pointed to by fsdPtr.
fsdPtr	Contains a pointer to an FSDRec containing the data to be copied to the file system descriptor.

SetFSInfo changes the file system descriptor specified by the fsid parameter. This routine is normally used following InstallFS to enable the foreign file system's component interfaces. bufSize bytes are copied to the file system descriptor from the buffer pointed to by fsdPtr. The fsdLink, fsdLength, fsdVersion, fileSystemFSID, fileSystemName, and fileSystemSpec fields in the file system descriptor are not modified by SetFSInfo.

Result codes

noErr	0	No error
fsmFFSNotFoundErr	-431	The file system descriptor with fsid was not found in the FSD queue
fsmBusyFFSErr	-432	An FSM component interfaces is busy; the file system descriptor cannot be modified
fsmBadFSDLenErr	-434	The number of bytes specified by bufSize is larger than the file system descriptor's fsdLength field
fsmNoAlternateStackErr	-437	An attempt was made to enable the file system's HFS component interface but no alternate stack was provided in fsdPtr->fsdHFSCI

InformFSM

Use the InformFSM function to communicate with the File System Manager.

```
pascal short InformFSM (short theMessage, void *paramBlock);
```

theMessage Contains the message to send to the File System Manager.

paramBlock Contains a pointer to optional message specific data.

InformFSM passes a message and optional message-specific data to the File System Manager. The File System Manager will return an fsmUnknownFSMMessageErr if an undefined message is passed. The currently defined messages are fsmNopMessage, fsmDrvQElChangedMessage, and fsmGetFSIconMessage. How the File System Manager handles each message and what result codes can be returned are described in the following paragraphs.

fsmNopMessage(0)

The File System Manager does nothing. The paramBlock parameter is ignored and the result is always noErr.

Result codes

noErr	0	No error

fsmDrvQElChangedMessage (1)

The fsmDrvQElChangedMessage tells the File System Manager to search the drive queue for unmounted volumes and attempt to find a File System Manager controlled foreign file system that can mount the volume. If a foreign file system is found that can mount the volume, the File System Manager will issue a diskInsertEvt for the drive so the foreign file system can mount the volume.

Note: Drive queue elements with the value fsmIgnoreFSID in the dQFSID field are ignored by the File System Manager when it searches for unmounted volumes.

This message can be sent by a disk driver to notify the File System Manager that a unmounted disk has a non-HFS partition. Before sending this message, the disk driver has allocated a drive queue element for the partition and added it to the system's drive queue.

This message is also sent by a foreign file system's installation code to check for unmounted disks after a foreign file system is installed.

The paramBlock parameter is ignored. The result is always noErr.

Note: This message can cause memory to move. A disk driver can safely make this request only at driver accRun time.

Result codes

noErr	0	No error

fsmGetFSIconMessage (2)

The fsmGetFSIconMessage tells the File System Manager to get a foreign file system's disk icon. The paramBlock parameter points to a FSMGetIconRec.

FSMGetIconRec

→	refNum	short	Contains the target drive number, volume reference number, or working directory number
→	iconBufferPtr	Ptr	Contains a pointer to the icon buffer where the foreign file system will return the icon data
→	requestSize	long	Contains the size of the supplied icon buffer
←	actualSize	long	The foreign file system returns the actual size of the icon data in this field
→	iconType	char	Contains the kind of icon requested
←	isEjectable	Boolean	The File System Manager returns true in this field if the device is ejectable
←	driveQElemPtr	DrvQElPtr	The File System Manager returns a pointer to the drive's DrvQEl in this field
←	fileSystemSpecPtr	FSSpecPtr	The File System Manager returns a pointer to foreign file system's FSSpec in this field
→	reserved1	long	reserved, must be zero

The refNum field must contain the target drive number, volume reference number or working directory number. The iconBufferPtr field must point to the buffer where the icon will be returned. The requestSize field must contain the size of the supplied icon buffer. The iconType field must contain the kind of icon requested.

The icon kinds supported and their sizes are those used by the Desktop Manager and are listed in Table 1-4.

Table 1-4. Icon Types

Constant	Value or bytes in bitmap	Corresponding resource type	Description
kLargeIcon	1	'ICN#'	Large black & white icon w/mask
kLarge4BitIcon	2	'icl4'	Large 4-bit color icon
kLarge8BitIcon	3	'icl8'	Large 8-bit color icon
kSmallIcon	4	'ics#'	Small black-and-white icon w/mask
kSmall4BitIcon	5	'ics4'	Small 4-bit color icon
kSmall8BitIcon	6	'ics8'	Small 8-bit color icon
kLargeIconSize	256	'ICN#'	Large black &white icon w/mask
kLarge4BitIconSize	512	'icl4'	Large 4-bit color icon
kLarge8BitIconSize	1024	'icl8'	Large 8-bit color icon
kSmallIconSize	64	'ics#'	Small black-and-white icon w/mask
kSmall4BitIconSize	128	'ics4'	Small 4-bit color icon
kSmall8BitIconSize	256	'ics8'	Small 8-bit color icon

The File System Manager uses the refNum field in the FSMGetIconRec to determine the drive and the foreign file system that owns the drive. Then, the File System Manager fills in the isEjectable, driveQElemPtr, and fileSystemSpecPtr fields in the FSMGetIconRec and passes the FSMGetIconRec to the foreign file system with a ffsGetIconMessage message.

The foreign file system is responsible for returning the icon in the buffer pointed to by iconBufferPtr and the icon data size in the actualSize field. See the description of the file system communications function's ffsGetIconMessage for more information.

Result codes

noErr	0	No error
nsvErr	-35	refNum in FSMGetIconRec did not specify a volume
nsDrvErr	-56	refNum in FSMGetIconRec did not specify a drive
afpItemNotFound	-5012	The foreign file system could not return the icon type requested
any of the result codes returned by FSpOpenResFile		

InformFFS

Use the InformFFS function to communicate with foreign file systems through the File System Manager.

```
pascal short InformFFS(short fsid, void *paramBlock);
```

fsid Contains the unique file system ID (FSID) of the foreign file system.

paramBlock Contains a pointer to optional message specific data.

InformFFS passes a message through the File System Manager to the foreign file system specified by fsid. The message data structure is defined by each foreign file system.

Result codes

noErr	0	No error
fsmFFSNotFoundErr	-431	The file system descriptor with fsid was not found in the FSD queue

FOREIGN FILE SYSTEM-DEFINED ROUTINES

File System Communications Function

The communications function pointed to by fileSystemCommProc in an FSDRec is used by the File System Manager to communicate with a foreign file system. The fileSystemCommProc must have the following form.

```
pascal OSErr MyFSDCommProc (short message, void *paramBlock,
                            void *globalsPtr);
```

message	Contains the message being sent to the foreign file system.
paramBlock	Contains a pointer to a buffer containing any additional parameters needed to process the message.
globalsPtr	Contains a pointer to the foreign file system's optional global data area.

When the file system communications function is called, it is passed a message and a pointer to a buffer containing any additional parameters needed to process the message. The currently defined messages are ffsNopMessage, ffsGetIconMessage, ffsIDDiskMessage, ffsLoadMessage, ffsUnloadMessage, ffsIDVolMountMessage, and ffsInformMessage. How the file system communications function should handle each message and what result codes should be returned are described in the following paragraphs. If the message passed is not recognized or handled by your file system communications function, the function result returned must be fsmUnknownFSMMessageErr (-438).

Note: The file system communications function is not called at interrupt time. Thus, it may make Memory Manager requests and synchronous operating system requests.

ffsNopMessage (0)

When the ffsNopMessage is passed to the file system communications function, do nothing and always return a result of noErr.

Result codes

noErr	0	No error

ffsGetIconMessage (1)

The ffsGetIconMessage is passed to the file system communications function when InformFSM is called with fsmGetFSIconMessage. The paramBlock parameter points to a FSMGetIconRec record.

FSMGetIconRec

→	refNum	short	Contains the target drive number, volume reference number, or working directory number
→	iconBufferPtr	Ptr	Contains a pointer to the icon buffer where the foreign file system will return the icon data
→	requestSize	long	Contains the size of the supplied icon buffer
←	actualSize	long	The foreign file system returns the actual size of the icon data in this field
→	iconType	char	Contains the kind of icon requested
→	isEjectable	Boolean	Contains true if the device is ejectable
→	driveQElemPtr	DrvQElPtr	Contains a pointer to the drive's DrvQEl
→	fileSystemSpecPtr	FSSpecPtr	Contains a pointer to foreign file system's FSSpec
→	reserved1	long	reserved, must be zero

Your foreign file system uses iconType to determine the type of icon requested. If a known icon type is requested, open the foreign file system's resource file (specified by fileSystemSpecPtr) with read-only access, get the specified icon resource, copy the icon data (up to requestSize bytes) into the buffer pointed to by iconBufferPtr, and then close the foreign file system's resource file. Return the number of icon data bytes copied into into the buffer in the actualSize field.

Result codes

noErr	0	No error
afpItemNotFound	-5012	The icon type requested could not be found
any of the result codes returned by FSpOpenResFile		

ffsIDDiskMessage(2)

The ffsIDDiskMessage is passed to the file system communications function when the File System Manager intercepts a MountVol request. If the idSector field in the foreign file system's HFSCIRec is not -1, the paramBlock parameter points to the disk block specified by the idSector field of the HFSCIRec. If the idSector field in the foreign file system's HFSCIRec is -1, the paramBlock parameter points to the parameter block used to make the MountVol request and the ioVRefNum field in that parameter block contains the drive number of the volume to mount.

Your foreign file system should determine if the disk volume belongs to this foreign file system and return noErr if it does. If the disk volume cannot be identified, return extFSErr.

If your foreign file system returns noErr and its HFS component interface code is not loaded, the File System Manager will send your foreign file system a ffsLoadMessage.

Result codes

noErr	0	The disk volume belongs to this foreign file system
extFSErr	-58	The disk volume does not belong to this foreign file system

ffsLoadMessage(3)

The ffsLoadMessage is passed to the file system communications function when the File System Manager needs to call the HFS component interface code of a foreign file system that is not loaded. The paramBlock parameter is not used.

You should check to see if the HFS component interface code is loaded. If not, load and detach the HFS component interface code and related data, allocate a stack for the HFS component interface code's use, initialize the HFSCIRec in the FSD, set the hfsCIResourceLoadedBit in the compInterfMask, activate the HFS component interface for the foreign file system, and return noErr.

If the HFS component interface code cannot be loaded, the stack cannot be allocated, or the HFS component interface for the foreign file system cannot be activated, return an error result.

Result codes

noErr	0	No error; the HFS component interface was successfully loaded and activated
memFullErr	-108	The stack could not be allocated

any of the result codes returned by the Resource Manager, GetFSInfo, or SetFSInfo

ffsUnloadMessage(4)

The ffsUnloadMessage is passed to the file system communications function when the File System Manager no longer needs the HFS component interface code in memory. The paramBlock parameter is not used.

You should disable the HFS component interface, clear the hfsCIResourceLoadedBit in the compInterfMask, make the HFS component interface code and related data purgable, and dispose of the HFS component interface code's stack.

Result codes

noErr	0	No error; the HFS component interface was successfully unloaded and deactivated

any of the result codes returned by GetFSInfo, or SetFSInfo

ffsIDVolMountMessage (5)

This message is passed to the file system communications function when the File Manager's VolumeMount routine is called to give a foreign file system a chance to claim the request. The paramBlock parameter points to parameter block used to make the VolumeMount request. The ioBuffer field of the parameter block points to a VolumeMountInfoHeader structure.

VolumeMountInfoHeader

→	length	short	Contains the length in bytes of the entire VolumeMountInfo record including this field and the variable length data following the flags field
→	media	VolumeType	Contains the VolumeType of the media to mount. This is the creator type that you registered with Apple Computer, Inc. for your file system ID number.
↔	flags	short	Contains the volume mount flags.
		→ volMountInteractBit	If set, it's safe for the file system to perform user interaction to mount the volume
		← volMountChangedBit	If set, the file system mounted the volume, but the volume mounting information record needs to be updated.

Note: The VolumeMountInfoHeader record is a superset of the VolMountInfoHeader record defined in Inside Macintosh: Files. In particular, the flags field used by the AppleShare file system was added to VolumeMountInfoHeader to allow foreign file systems to properly interact with the Alias Manager (the kARMNoUI rulesMask passed to MatchAlias is used to set or clear the volMountInteractBit; MatchAlias uses the volMountChangedBit to determine the value returned in the needsUpdate parameter).

You should compare the media field in the VolumeMountInfoHeader to your foreign file system's media type and return noEr if they match. If they do not match, return extFSErr. Your media type should be the creator type assigned when you register your foreign file system's FSID.

If your foreign file system returns noErr and its HFS component interface code is not loaded, the File System Manager will send your foreign file system a ffsLoadMessage.

Foreign file systems are called with the ffsIDVolMountMessage before the VolumeMount request is passed to the File Manager. If your foreign file system claims the request, it should use this time to load data or code resources and to perform any user interaction needed to mount the volume. User interface interactions should be performed only if the volMountInteract bit is set in the flags field. When the volMountInteract bit is set, it indicates the caller of VolumeMount has initialized the QuickDraw, the Font Manager, the Window Manager, the Menu Manager, TextEdit, and the Dialog Manager.

If the volume can be mounted and the foreign file system detemines the volume mounting information record needs to be updated, the foriegn file system should set the volMountChangedBit to indicate to the caller that it should call GetVolMountInfoSize and GetVolMountInfo to get an updated version of the volume mounting information record.

Note: When the foreign file system's HFSCIProc is later called with the VolumeMount request, the foreign file system cannot call any system routines which might cause a File Manager request because the File Manager is single-threaded and not reentrant.

Result codes

noErr	0	The VolumeMount request belongs to this foreign file system
extFSErr	-58	The VolumeMount request does not belong to this foreign file system

ffsInformMessage(6)

This message is passed to the file system communications function when InformFFS is called. The paramBlock parameter is the same paramBlock parameter passed to InformFFS.

The ffsInformMessage allows other programs to pass your foreign file system messages defined by you.

Result codes

noErr	0	No error
any result codes defined by your foreign file system for this request		

2 THE HFS COMPONENT INTERFACE

ABOUT THIS CHAPTER

This chapter describes the HFS component interface, the data structures used by the HFS component interface, and the routines your foreign file system must provide to the HFS component interface.

To use this chapter, you should be familiar with the information in the chapters "The File System Manager" and "File System Utility Routines" in this document, the information in *Inside Macintosh: Files*, and the information in *Inside Macintosh: Devices*.

ABOUT THE HFS COMPONENT INTERFACE

The HFS component interface allows a foreign file system to process File Manager requests and allows a foreign file system to describe its file system services to the File System Manager. Whenever a File Manager request is made to a volume controlled by a foreign file system, the File System Manager will pass that request to the foreign file system through the HFS component interface.

USING THE HFS COMPONENT INTERFACE

This section describes

- the HFS Component Interface record
- the foreign file system's stack
- the HFS Component Interface request processing function
- the Logical to Physical function
- the Extend File function

The HFS Component Interface Record

The HFS component interface record in an File System Descriptor lets the foreign file system tell the HFS component interface how to dispatch file system requests to the foreign file system. It tells the HFS component interface where the routines to handle file system requests are in memory, what disk block the foreign file system needs to identify a disk, and where the foreign file system's stack is. The data type HFSCIRec defines the HFS component interface record.

```
struct HFSCIRec {
     long               compInterfMask;    /* component flags */
     HFSCIUPP           compInterfProc;    /* pointer to file system
                                              request processing code */
     Lg2PhysUPP         log2PhyProc;       /* pointer to Lg2PhysProc */
     Ptr                stackTop;          /* file system stack top */
     long               stackSize;         /* file system stack
                                              size */
     Ptr                stackPtr;          /* current file system
                                              stack pointer */
     long               reserved3;         /* --reserved, must be
                                              zero-- */
     long               idSector;          /* Sector you need to ID a
                                              local volume. For
                                              networked volumes, this
                                              should be -1. */
     long               reserved2;         /* --reserved, must be
                                              zero-- */
     long               reserved1;         /* --reserved, must be
                                              zero-- */
     };
     typedef struct HFSCIRec HFSCIRec;
     typedef HFSCIRec *HFSCIRecPtr;
```

Field descriptions

compInterfMask Contains the HFS component interface dispatch mask. Currently the following bits are defined:

Bit		Meaning
0-17		Reserved.
18	hfsCIHLL2PProcBit	Set this bit if the log2PhyProc is written in a high level language using Pascal calling conventions. Note: this bit should always be set by foreign file systems using the logical to physical interface described in this chapter.
19	hfsCIResourceLoadedBit	Set this bit if the HFSCIProc code resource is loaded. If the foreign file system doesn't support dynamic loading, this bit should always be set.
20	hfsDoesDynamicLoadBit	Set this bit if dynamically loading code resource is supported.
21	hfsCIDoesDeskTopBit	Set this bit if Desktop Manager requests are supported. Obsolete, but should be set. Current versions of FSM always pass Desktop Manager requests on to foreign file systems.

22	hfsCIDoesAppleShareBit	Set this bit if AppleShare requests are supported. Obsolete, but should be set. Current versions of FSM always pass AppleShare requests on to foreign file systems.
23	hfsCIDoesHFSBit	Set this bit if HFS requests are supported. Obsolete, but should be set. Current versions of FSM always pass HFS requests on to foreign file systems.
24-29		Reserved for the File System Manager's use.
30	fsmComponentBusyBit	If set, the FSM component interface is busy (i.e., one or more requests are outstanding). This bit is maintained by the component and used by the File System Manager to control the use of the SetFSInfo File System Manager service routine.
31	fsmComponentEnableBit	If set, the component may begin dispatching requests to the foreign file system. The foreign file system should set this bit with the SetFSInfo File System Manager service routine once it is ready to receive requests.

compInterfProc — Contains a pointer to the foreign file system's HFS component interface request processing function .

log2PhyProc — Contains a pointer to the foreign file system's Logical to Physical routine.

stackTop — Contains a pointer to the top of the foreign file systems's stack. stackTop must be word aligned value since it becomes the stack pointer whenever the foreign file systems is called.

stackSize — Contains the size of the foreign file systems's stack.

stackPtr — Reserved for future use by the File System Manager.

reserved3 — Reserved, must contain zero.

idSector — Contains the sector the foreign file system needs to identify a local volume, or contains -1 if a network foreign file system. When the file system communications function is called with ffsIDDiskMessage and idSector is not -1, the paramBlock parameter points to the disk block specified. When the file system communications function is called with ffsIDDiskMessage and idSector is -1, the paramBlock parameter points to the parameter block used to make the MountVol request and the ioVRefNum field in that parameter block contains the drive number of the volume to mount.

reserved2 — Reserved, must contain zero.

reserved1 — Reserved, must contain zero.

The Foreign File System's Stack

Each foreign file system must have its own alternate stack. The File System Manager's HFS component interface saves the original application stack and switches to the foreign file system's alternate stack before calling the foreign file system's request processing function. After the foreign file system services the File Manager request, The File System Manager's HFS component interface switches back to the original application stack.

If the foreign file system performs I/O through the File System Utility cached I/O routines, there will be an additional stack switch while the HFS file system handles the I/O. Foreign file system Logical to Physical routines run on the foreign file system's stack.

The foreign file system's alternate stack must be allocated in the system heap with NewPtrSys before the HFS component interface is activated. If the foreign file system supports dynamically loading code resources, the stack can be allocated when the file system communications function is called with the ffsLoadMessage and released when the file system communications function is called with the ffsUnloadMessage.

The size of a foreign file system's alternate stack should be the amount needed by your foreign file system plus two or three kilobytes of additional space for use by interrupt driven processes which may borrow space at interrupt time from your stack.

The HFS Component Interface Request Processing Function

The HFS Component Interface request processing function pointed to by compInterfProc in a HFSCIRec is called by the File System Manager to handle File Manager requests. The HFS Component Interface request processing function must have the following form.

```
pascal OSErr HFSCIProc(VCBPtr theVCB, short selectCode,
                       void *paramBlock, void *fsdGlobalPtr,
                       short fsid);
```

theVCB	Contains a pointer to the VCB of the volume that is the target of the file system request.
selectCode	Contains either the A-Trap number or the HFSDispatch selector number which indicates what file system request was made.
paramBlock	Contains a pointer to the parameter block passed to the file system request.
fsdGlobalPtr	Contains a pointer to the foreign file system's optional global data area.
fsid	Contains the file system ID number of the volume.

When a foreign file system's HFSCIProc is called, it is told which volume (theVCB) is the target of the file system request (except for MountVol and VolumeMount as explained later in this section), what file system request was made, and is passed the parameter block used to make the file system request. The foreign file system should handle the request, fill in the appropriate fields in the parameter block, and return the appropriate result. The File System Manager provides File System Utility routines to your foreign file system to help it process file system request.

▲ **Warning:** When the foreign file system's HFSCIProc is called, the foreign file system cannot do anything which might cause another File Manager request because the File Manager is single-threaded and is not reentrant. If your foreign file system's HFSCIProc does cause another File Manager request, the Macintosh system will deadlock. ▲

The HFSCIProc is also passed a pointer to the foreign file system's global data area and the file system ID number of the volume. The file system ID number allows a foreign file system that handles multiple file systems to quickly determine which file system needs to handle the file system request.

Result codes

noErr	0	The foreign file system handled the Macintosh file system request with no errors

any of the result codes documented for a particular Macintosh file system request in *Inside Macintosh: Files*

Listing 2-1 is an abbreviated example of a request processing function. Not all HFSDispatch selectors and Macintosh file system A-Traps are shown in this example.

Listing 2-1. HFS Component Interface request processing function

```
/* Prototype for file system request handler routine */
typedef OSErr (*hfsProcPtr)(VCBPtr theVCB, short selectCode,
                            void *paramBlock, void *fsdGlobalPtr,
                            short fsid);

pascal OSErr   HFSCIProc (VCBPtr theVCB, short selectCode,
                          void *paramBlock, void *fsdGlobalPtr,
                          short fsid)
{
   OSErr       result = extFSErr;
   hfsProcPtr     hfsProc;
   unsigned short trapWord;

   /* selectCode contains the trap word or HFSDispatch selector */
   /* Clear the trap attributes (i.e., async and immediate bits) */
   trapWord = selectCode & 0xF0FF;

   /* and find out what to call with the big, big switch statement... */
   switch (trapWord)
   {
```

```
   /* HFSDispatch (A260) file system selectors */
   case kFSMOpenWD:
      hfsProc = DoOpenWD;
      break;
   /*
   ** The rest of HFSDispatch (A260) file system selectors go here.
   */
   case kFSMMakeFSSpec:
      hfsProc = DoMakeFSSpec;
      break;

   /* HFSDispatch (A260) Desktop Manager selectors */
   case kFSMDTGetPath:
      hfsProc = DoDTGetPath;
      break;
   /*
   ** The rest of HFSDispatch (A260) Desktop Manager selectors
   ** go here.
   */
   case kFSMDTDelete:
      hfsProc = DoDTDelete;
      break;

   /* HFSDispatch (A260) AppleShare selectors */
   case kFSMGetVolParms:
      hfsProc = DoGetVolParms;
      break;
   /*
   ** The rest of HFSDispatch (A260) AppleShare selectors
   ** go here.
   */
   case kFSMSetForeignPrivs:
      hfsProc = DoSetForeignPrivs;
      break;

   /* File system A-traps */
   case kFSMOpen:
      hfsProc = DoOpen;
      break;
   /*
   ** The rest of File system A-traps go here.
   */
   case kFSMFlushFile:
      hfsProc = DoFlushFile;
      break;

   /* and if it wasn't there... */
   default:
      hfsProc = DoRequestNotSupported;
      break;
}
```

```
   /* Call the appropriate file system request handler routine */
   result = hfsProc(theVCB, selectCode, paramBlock, fsdGlobalPtr, fsid);
   return(result);
}
```

Handling MountVol and VolumeMount requests

There are two File Manager requests sent to a foreign file system's HFSCIProc that may or may not be handled by your foreign file system, MountVol (selectCode = $A00F) and VolumeMount (selectCode = $0041). The File System Manager sends those requests to each installed foreign file system until one of the foreign file systems indicates that it has handled the request.

If your foreign file system receives a MountVol request, it must determine if the disk volume belongs to it. You may be able to use the same code your foreign file system used to handle the ffsIDDiskMessage is passed to the file system communications function. If the volume is not your volume, you must return extFSErr. If the volume is your volume, you must return noErr if you successfully mount the volume, or one of the error result codes listed for PBMountVol in Inside Macintosh: Files.

If your foreign file system receives a VolumeMount request and supports the VolumeMount related selectors (VolumeMount, GetVolMountInfoSize, and GetVolMountInfo), it should compare the media field in the VolumeMountInfoHeader to your foreign file system's media type. If the media field does not match your media type, return extFSErr. If the media field matches your media type, you must return noErr if you successfully mount the volume, or one of the error result codes listed for PBVolumeMount in Inside Macintosh: Files. If your foreign file system does not support the VolumeMount related selectors, it should return paramErr.

The Logical to Physical Block Mapping Function

The Macintosh cache routines described in "File System Utility Routines" chapter can read or write either logical blocks of a file or physical blocks of a volume. However, because disk driver device requests always read or write physical blocks of a volume, a foreign file system must provide a routine to map logical file blocks to physical volume blocks. That routine is called the logical to physical routine. The address of the logical to physical routine is passed to the File System Manager in the log2PhyProc field of the foreign file system's HFSCIRec and as a parameter to some cache routines. The logical to physical routine must have the following form.

```
pascal OSErr Lg2PhysProc(void *fsdGlobalPtr,
                         VCBPtr volCtrlBlockPtr,
                         FCBRecPtr fileCtrlBlockPtr,
                         short fileRefNum,
                         unsigned long filePosition,
                         unsigned long reqCount,
                         unsigned long *volOffset,
                         unsigned long *contiguousBytes);
```

fsdGlobalPtr	Contains a pointer to the foreign file system's optional global data area.
volCtrlBlockPtr	Contains a pointer to the file's Volume Control Block (VCB).
fileCtrlBlockPtr	Contains a pointer to the file's File Control Block (FCBRec) that is the target of the logical to physical mapping.
fileRefNum	Contains the file's reference number.
filePosition	Contains the byte offset into the file's data where the mapping should begin (always a multiple of 512 bytes).
reqCount	Contains the number of bytes requested for the mapping operating (always a multiple of 512 bytes).
volOffset	Contains a pointer to an unsigned long. Lg2PhysProc places the offset (in bytes) to the volume block where the file data can be found into the field referred to by this parameter. The value returned must be a multiple of 512 bytes.
contiguousBytes	Contains a pointer to an unsigned long. Lg2PhysProc places the number of contiguous bytes which occur after the given offset, up to requestCount, into the field referred to by this parameter. The value returned must be a multiple of 512 bytes.

It is the responsibility of the foreign file system to return an eofErr when a request for data is beyond the current logical end of file (before calling the Cache utility routines). It is the responsibility of the logical to physical routine to return an eofErr error when a cache routine asks for block mapping beyond the current physical end of file.

Result codes		
noErr	0	Logical to physical mapping was successful
eofErr	-39	Requested mapping was beyond the physical end of file
ioErr	-36	Mapping failed because of an unexpected error

Here is a brief description of how logical to physical mapping works. Assume that a file is open and that file occupies the physical blocks 101 through 103 and 105 through 109 (8 disk blocks) on the disk volume. The logical to physical routine needs to map bytes 0 through 1535 to blocks 101 through 103 and map bytes 1536 through 8191 to blocks 105 through 109. Here's an example of how a logical to physical routine would perform the job of mapping a read to the fragmented file.

A Read request is recieved through the HFSCIProc asking for 3072 bytes starting at offset 0 in the file. The HFSCIProc calls the file system's Read request hander routine which eventually calls UTCacheReadIP to read the data from the file. The first time UTCacheReadIP is called, the Macintosh cache mechanism calls your logical to physical routine with filePosition = 0 and reqCount = 3072. The logical to physical routine returns 51712 in volOffset (the byte offset of block 101) and 1536 in contiguousBytes (the number of contiguous bytes starting at offset 51712). The Macintosh cache mechanism will then read 1536 bytes starting at offset 51200 into the buffer passed to UTCacheReadIP.

Since the Read request hasn’t been satisfied, the file system’s Read request hander routine updates the buffer pointer and calls UTCacheReadIP again - this time asking for 1536 bytes starting at offset 1536 in the file. The Macintosh cache mechanism calls your logical to physical routine with filePosition = 1536 and reqCount = 1536. The logical to physical routine returns 53760 in volOffset (the byte offset of block 105) and 1536 in contiguousBytes (the number of contiguous bytes starting at offset 53760). The Macintosh cache mechanism will then read 1536 bytes starting at offset 53760 into the buffer passed to UTCacheReadIP and the Read request is complete.

3 THE FILE SYSTEM UTILITY ROUTINES

ABOUT THIS CHAPTER

This chapter describes the utility routines provided by the File System Manager to your foreign file system to help it process Macintosh file system calls and describes the data structures used by the Macintosh file system.

To use this chapter, you should be familiar with the information in the chapter "The HFS Component Interface" in this document, the information in *Inside Macintosh: Files*, and the information in *Inside Macintosh: Devices*.

ABOUT FILE SYSTEM UTILITY ROUTINES

The File System Utility routines are intended for use by foreign file systems to allow your foreign file system to

- access file control blocks (FCBs), volume control blocks (VCBs), working directory control blocks (WDCBs), and drive queue elements (DrvQEl)
- set and get the default volume and working directory
- eject a volume
- validate and process parameters passed with Macintosh file system calls
- access the Macintosh file system's cache buffers and low-level I/O services

▲ **Warning:** File System Utility routines that can cause I/O through the Macintosh file system's cache, UTGetBlock, UTReleaseBlock, UTFlushCache, UTCacheReadIP, and UTCacheWriteIP, must be called from a foreign file system handling a request through its HFSCIProc. Attempts to call those routines outside of the context of the HFSCIProc will cause a system crash. ▲

MACINTOSH FILE SYSTEM DATA STRUCTURES

This section describes the data structures used internally by the Macintosh file system. Foreign file systems need to access and manipulate these data strutures when handling calls from the HFS component interface.

The data structures include

- volume control blocks (VCBs)
- file control blocks (FCBs)

- working directory control blocks (WDCBs)
- drive queue elements (DrvQEl)

The data structures (except for working directory control blocks) and their use are described in *Inside Macintosh: Files*. The following sections describe how foreign file systems should use them.

Volume Control Blocks

Each time a volume is mounted, a foreign file system must allocate a volume control block (VCB) with UTAllocateVCB, read the volume and use the information to initialize the fields in the new VCB, and add the VCB to the VCB queue with UTAddNewVCB (unless an ejected or offline volume is being remounted). When a volume is unmounted, its VCB is removed from the VCB queue UTDisposeVCB.

The volume control block queue is a standard Operating System queue maintained in the system heap. It contains a volume control block for each mounted volume. A volume control block is a nonrelocatable block that contains volume-specific information. If your foreign file system needs to keep additional information beyond that kept in the system VCB structure, you can allocate additional memory with UTAllocateVCB for your extended VCB structure. The structure of a system volume control block is defined by the VCB data type.

```
struct VCB {                                /* volume control block */
     QElemPtr         qLink;                /* next queue entry */
     short            qType;                /* queue type */
     short            vcbFlags;             /* volume flags */
     unsigned short   vcbSigWord;           /* volume signature */
     unsigned long    vcbCrDate;            /* date and time of volume
                                               creation */
     unsigned long    vcbLsMod;             /* date and time of last
                                               modification
     short            vcbAtrb;              /* volume attributes */
     unsigned short   vcbNmFls;             /* number of files in root
                                               directory */
     short            vcbVBMSt;             /* first block of volume
                                               bitmap */
     short            vcbAllocPtr;          /* start of next allocation
                                               search */
     unsigned short   vcbNmAlBlks;          /* number of allocation blocks
                                               in volume */
     unsigned long    vcbAlBlkSiz;          /* size (in bytes) of
                                               allocation blocks
     unsigned long    vcbClpSiz;            /* default clump size */
     short            vcbAlBlSt;            /* first allocation block in
                                               volume */
     long             vcbNxtCNID;           /* next unused catalog node
                                               ID */
     unsigned short   vcbFreeBks;           /* number of unused allocation
                                               blocks */
     Str27            vcbVN;                /* volume name */
```

```
short            vcbDrvNum;        /* drive number */
short            vcbDRefNum;       /* driver reference number */
short            vcbFSID;          /* file-system identifier */
short            vcbVRefNum;       /* volume reference number */
Ptr              vcbMAdr;          /* used internally */
Ptr              vcbBufAdr;        /* used internally */
short            vcbMLen;          /* used internally */
short            vcbDirIndex;      /* used internally */
short            vcbDirBlk;        /* used internally */
unsigned long    vcbVolBkUp;       /* date and time of last
                                      backup */
unsigned short   vcbVSeqNum;       /* volume backup sequence
                                      number */
long             vcbWrCnt;         /* volume write count */
long             vcbXTClpSiz;      /* clump size for extents
                                      overflow file */
long             vcbCTClpSiz;      /* clump size for catalog
                                      file */
unsigned short   vcbNmRtDirs;      /* number of directories in
                                      root dir */
long             vcbFilCnt;        /* number of files in
                                      volume */
long             vcbDirCnt;        /* number of directories in
                                      volume */
long             vcbFndrInfo[8];   /* information used by the
                                      Finder */
unsigned short   vcbVCSize;        /* used internally */
unsigned short   vcbVBMCSiz;       /* used internally */
unsigned short   vcbCtlCSiz;       /* used internally */
unsigned short   vcbXTAlBlks;      /* size of extents overflow
                                      file */
unsigned short   vcbCTAlBlks;      /* size of catalog file */
short            vcbXTRef;         /* ref. num. for extents
                                      overflow file */
short            vcbCTRef;         /* ref. num. for catalog
                                      file */
Ptr              vcbCtlBuf;        /* ptr. to extents and catalog
                                      caches */
long             vcbDirIDM;        /* directory last searched */
short            vcbOffsM;         /* offspring index at last
                                      search */

};
typedef struct VCB VCB;
typedef VCB *VCBPtr;
```

Field descriptions

qLink — A pointer to the next entry in the VCB queue. This field is initialized when the VCB is added to the VCB queue by UTAddNewVCB.

qType — The queue type. This field is initialized when the VCB is added to the VCB queue by UTAddNewVCB.

vcbFlags — Volume flags. Set bit 15 if the volume information has been changed by a File Manager call since the volume was last affected by a FlushVol call.

vcbSigWord — The volume signature. This field must be set to $4244.

vcbCrDate — Set to the date and time of volume creation (initialization), specified as the number of seconds since midnight, January 1, 1904.

vcbLsMod — Set to the date and time of last modification, specified as the number of seconds since midnight, January 1, 1904. This is not necessarily when the volume was last flushed.

vcbAtrb — Volume attributes. The bits have these meanings:

Bit	Meaning
0–5	Reserved.
6	Set this bit if the volume is busy (one or more files are open).
7	Set this bit if the volume is locked by hardware. This can be obtained when the volume is mounted with the DriveStatus function.
8–14	Reserved.
15	Set this bit if the volume is locked by software. This bit can be set or cleared by programs calling your foreign file system with _SetVInfo and you should store it between volume mounts.

vcbNmFls — Set to the number of files in the volume's root directory.

vcbVBMSt — For HFS volumes, this field is used to point to the first block of the volume bitmap. Your foreign file system can use this field to point to an analogous structure. If it doesn't, the field's value must be 0.

vcbAllocPtr — For HFS volumes, this field is used to point to the start block of the next allocation search. Your foreign file system can use this field for a similar purpose. If it doesn't, the field's value must be 0.

vcbNmAlBlks — Set to the number of allocation blocks in the volume (maximum $FFFF).This field, together with the vcbAlBlkSiz field, is used to determine the number of bytes on a particular volume. It must be initialized to a value which when multiplied with vcbAlBlkSiz provides a reasonable estimate of the size of the disk.

vcbAlBlkSiz — Set to the allocation block size (in bytes). This field, together with the vcbNmAlBlks field, is used by the Finder and other applications to determine the number of bytes on a particular volume. It must be initialized to some non-zero multiple of 512.

vcbClpSiz — Set to the default clump size. For HFS volumes, this field is used to determine how many new allocation blocks to allocate when the file is extended. Your foreign file system can use this field for the same purpose, but in any case, it must set its value to a multiple of vcbAlBlkSiz or 0.

vcbAlBlSt	Set to the first allocation block in the volume. For HFS volumes, this is the number of the first block used for files. If your foreign file system has an analogous structure (that is, if its header blocks are followed by a contiguous set of blocks for files), it can use this field in the same way. If not, it must leave this field zero.
vcbNxtCNID	The next unused catalog node ID (directory ID or file ID). For HFS volumes, this is set to fsUsrCNID when the volume is initialized. When a new directory or file is created, vcbNxtCNID is used as the directory ID or file ID and then incremented. Your foreign file system can use this field for the same purpose, or may leave it zero.
vcbFreeBks	Set to the number of unused allocation blocks on the volume (maximum $FFFF). This field, together with vcbAlBlkSiz, is used by the Finder and other applications to determine the number of free bytes in the volume. It must be initialized to a value which when multiplied with vcbAlBlkSiz provides a reasonable estimate of the free space in the volume.
vcbVN	Set to the volume name. This field consists of a length byte followed by 27 bytes. Note that the volume name can occupy at most 27 characters; this is an exception to the normal file and directory name limit of 31 characters. The volume name must conform to the rules for HFS volumes (i.e., no colon characters). For file systems that have no volume-naming conventions, some standard name (such as “Untitled”) must be provided.
vcbDrvNum	The drive number of the drive on which the volume is located. This field is initialized when the VCB is added to the VCB queue by UTAddNewVCB. When a mounted volume is placed offline or ejected, vcbDrvNum is set to 0.
vcbDRefNum	The driver reference number of the driver used to access the volume. This field is initialized when the VCB is added to the VCB queue by UTAddNewVCB. When a volume is ejected, vcbDRefNum is set to the previous value of vcbDrvNum (and hence is a positive number). When a volume is placed offline, vcbDRefNum is set to the negative of the previous value of vcbDrvNum (and hence is a negative number).
vcbFSID	An identifier for the file system handling the volume; it is zero for volumes handled by the File Manager and nonzero for volumes handled by other file systems. Set to your foreign file system’s file system ID.
vcbVRefNum	The volume reference number. This field is initialized when the VCB is added to the VCB queue by UTAddNewVCB.
vcbMAdr	Used internally. Reserved.
vcbBufAdr	Used internally. Reserved.
vcbMLen	Used internally. Reserved.
vcbDirIndex	Used internally. Reserved.
vcbDirBlk	Used internally. Reserved.

vcbVolBkUp	Set to the date and time of the last volume backup, specified as the number of seconds since midnight, January 1, 1904.
vcbVSeqNum	Used internally. Reserved.
vcbWrCnt	The volume write count. This field is incremented every time the volume is written to by the Macintosh file system's cache. If your foreign file system uses the Macintosh file system's cache, it must either initialize this field to zero when a volume is mounted, or it must initialize this field to zero the first time the volume is mounted and then save the current value of this field when a volume is unmounted and restore the value when the volume is mounted again to keep a running count of writes to the volume.
vcbXTClpSiz	The clump size of the extents overflow file. Reserved.
vcbCTClpSiz	The clump size of the catalog file. Reserved.
vcbNmRtDirs	Set to the number of directories in the root directory or leave it zero.
vcbFilCnt	Set to the number of files on the volume or leave it zero.
vcbDirCnt	Set to the number of directories on the volume (not including the root directory) or leave it zero.
vcbFndrInfo	Information used by the Finder. For HFS volumes, this field is used internally by HFS. Leave it all zeroes until set by _SetVInfo.
vcbVCSize	Used internally. Reserved.
vcbVBMCSiz	Used internally. Reserved.
vcbCtlCSiz	Used internally. Reserved.
vcbXTAlBks	The size (in blocks) of the extents overflow file. Reserved.
vcbCTAlBks	The size (in blocks) of the catalog file. Reserved.
vcbXTRef	The path reference number for the extents overflow file. Reserved.
vcbCTRef	The path reference number for the catalog file. Reserved.
vcbCtlBuf	A pointer to the extents and catalog caches. Reserved.
vcbDirIDM	The directory last searched. For HFS volumes, this field is used internally by HFS to optimize GetCatInfo and GetFInfo. Your foreign file system can use this field for the same purpose, or may leave it zero.
vcbOffsM	The offspring index at the last search. For HFS volumes, this field is used internally by HFS to optimize GetCatInfo and GetFInfo. Your foreign file system can use this field for the same purpose, or may leave it zero.

There are also three File System Utility routines that let you search the VCB queue

- UTLocateVCBByRefNum searches for the VCB using a working directory number, volume reference number, or drive number

- UTLocateVCBByName searches for the first VCB with a volume name
- UTLocateNextVCB searches the VCB queue for the next VCB with the volume name passed to UTLocateVCBByName (the Macintosh file system allows multiple mounted volumes to have the same volume name)

Maximum Volume and File Sizes

The maximum volume size supported by the current File Manager programming interface is slightly less than 4 Gigabytes. The exact number is the maximum allocation block size ($FE00) multiplied by the maximum number of allocation blocks ($FFFF) — that's $FDFF0200 or 4261347840 decimal. To access volumes 2 Gigabytes or larger (larger than $7FFFFFFF or 2147483647 decimal), your foreign file system must used unsigned longword values when determining volume offsets and disk drivers called by your foreign file system must assume that the position passed to Read and Write requests in ioPosOffset is an unsigned longword value.

However, many applications, including versions of the Macintosh Finder, use signed longword values when determining volume size and volume freespace. Because of this, volumes 2 Gigabytes or larger may cause unexpected behavior.

Starting with System 7.5, the Macintosh Finder has been changed to use unsigned longword values when determining volume size and volume freespace. The File Manager routine PBHGetVInfo has also been changed starting with System 7.5 to pin the number of allocation blocks (vcbNmAlBlks) and the number of free allocation blocks (vcbNmAlBlks) so that when multiplied by the allocation block size (vcbAlBlkSiz), the result will be less than 2 Gigabytes. This prevents unexpected behavior from other applications that use signed longwords when determining volume size and volume freespace. The values in the VCB fields vcbNmAlBlks and vcbNmAlBlks are not changed. Listing 3-1 illustrates how you use Gestalt to determine if 4 Gigabyte volumes are supported by the Macintosh operating system.

Listing 3-1. Testing for 4 Gigabyte volume support

```
Boolean  Has4GBVols(void)
{
   long response;

   if (Gestalt(gestaltFSAttr, &response) == noErr)
      return ((response & (1L << gestaltFSSupports4GBVols)) != 0);
   else
      return (false);
}
```

The maximum file size supported by the current File Manager programming interface is 2 Gigabytes - 1 byte ($7FFFFFFF or 2147483647 decimal) because File Manger Read and Write requests use ioPosOffset as a signed longword value.

File Control Blocks

Each time a file is opened on a volume owned by a foreign file system, the foreign file system must read that file's catalog entry and build a file control block (FCB) in the FCB array. A new FCB is obtained with the UTAllocateFCB function.

When a file is closed, the foreign file system must mark the FCB free and release it with the UTReleaseFCB function.

The structure of a file control block is defined by the FCBRec data type.

```
struct FCBRec {
     unsigned long    fcbFlNm;          /* FCB file number. Non-zero
                                           marks FCB used */
     SignedByte       fcbFlags;         /* FCB flags */
     SignedByte       fcbTypByt;        /* File type byte */
     unsigned short   fcbSBlk;          /* File start block (in
                                           allocation size blocks) */
     unsigned long    fcbEOF;           /* Logical length or EOF in
                                           bytes */
     unsigned long    fcbPLen;          /* Physical file length in
                                           bytes */
     unsigned long    fcbCrPs;          /* Current position within
                                           file */
     VCBPtr           fcbVPtr;          /* Pointer to the
                                           corresponding VCB */
     Ptr              fcbBfAdr;         /* File's buffer address */
     unsigned short   fcbFlPos;         /* Directory block this file
                                           is in */
     unsigned long    fcbClmpSize;      /* Number of bytes per
                                           clump */
     Ptr              fcbBTCBPtr;       /* Pointer to B*-Tree control
                                           block for this file */
     unsigned long    fcbExtRec[3];     /* First 3 file extents */
     OSType           fcbFType;         /* File's 4 Finder Type
                                           bytes */
     unsigned long    fcbCatPos;        /* Catalog hint for use on
                                           Close */
     unsigned long    fcbDirID;         /* Parent Directory ID */
     Str31            fcbCName;         /* CName of open file */
     };
     typedef struct FCBRec FCBRec;
     typedef FCBRec *FCBRecPtr;
```

Field descriptions

fcbFlNum — Set to the file ID of this file. File numbers less than fsUsrCNID are considered system files.

fcbFlags — Flags describing the status of the file. Currently the following bits are defined:

Bit		Meaning
0	fcbWriteBit	Set if data can be written to the file
1	fcbResourceBit	Set if this FCB describes a resource fork
2	fcbWriteLockedBit	Set if the file has a locked byte range
3		Reserved
4	fcbSharedWriteBit	Set if the file has shared write permissions
5	fcbFileLockedBit	Set if the file is locked (write-protected)
6	fcbOwnClumpBit	Set if the file's clump size is specified in the FCB
7	fcbModifiedBit	Set if the file has changed since it was last flushed

fcbSBlk — Used internally for volumes owned by the Macintosh file system. Your foreign file system can use this field for its own purposes.

fcbEOF — Set to the logical end-of-file of the file.

fcbPLen — Set to the physical end-of-file of the file.

fcbCrPs — Set to the position of the mark.

fcbVPtr — Set to point to the volume control block of the volume containing the file.

fcbBfAdr — Reserved for internal use by the file system.

fcbFlPos — Used internally for volumes owned by the Macintosh file system. Your foreign file system can use this field for its own purposes.

fcbClmpSize — Set to the clump size of the file.

fcbBTCBPtr — Reserved for internal use by the file system.

fcbExtRec — For HFS volumes, this is an extent record (12 bytes) containing the first three extents of the file. Your foreign file system can use this field for its own purposes.

fcbFType — Set to the file's Finder type.

fcbCatPos — For HFS volumes, a catalog hint, used when the file is closed. Your foreign file system can use this field for its own purposes.

fcbDirID — The file's parent directory ID number.

fcbCName — The file's name. This field consists of a length byte followed by 31 bytes. The file name must conform to the rules for HFS file names (i.e., no colon characters). For file systems that have no file-naming conventions, some standard name (such as "Untitled") must be provided.

There are also four File System Utility routines that let you search the FCB array

- UTLocateFCB lets you find a FCB on a particular volume by file number or by file name

- UTLocateNextFCB resumes searching for a FCB by file number or file name from the FCB found by UTLocateFCB (the same file can have multiple access paths)
- UTIndexFCB lets you find, one at a time, all of the FCBs on a particular volume
- UTResolveFCB returns a pointer to a FCBRec given a file reference number

Note: If your foreign file system attempts to allocate a FCB in response to an _Open call and UTAllocateFCB cannot find a free FCB in the FCB array, you should return the tmfoErr result from UTAllocateFCB as your result. The Macintosh file system will then attempt to enlarge the FCB array and if successful, will retry the _Open call. Because enlarging the FCB array will change the array's location in memory, you should never save the location of a FCBRec between calls to your foreign file system. A FCBRecPtr can only be used within the context of one file system operation.

Working Directory Control Blocks

Working directories allow applications that use the old MFS File Manager routines to work on volumes with directory trees. Working directories allow the use of a volume reference number to specify both a volume and a directory on the volume. Each time a working directory is opened on a volume owned by a foreign file system, the foreign file system must build a working directory control block (WDCB) in the WDCB array with the UTAllocateWDCB function.

When a working directory is closed, the foreign file system must mark the WDCB free and release it with the UTReleaseWDCB function. When a volume is unmounted and its VCB is disposed of, UTDisposeVCB closes all working directories open on the volume.

The structure of a working directory control block is defined by the WDCBRec data type.

```
struct WDCBRec {
     VCBPtr           wdVCBPtr;        /* Pointer to VCB of this
                                          working directory */
     long             wdDirID;         /* Directory ID number of this
                                          working directory */
     long             wdCatHint;       /* Hint for finding this
                                          working directory */
     long             wdProcID;        /* Process that created this
                                          working directory */
     };
     typedef struct WDCBRec WDCBRec;
     typedef WDCBRec *WDCBRecPtr;
```

Field descriptions

wdVCBPtr	A pointer to the VCB of the volume this working directory refers to.
wdDirID	The directory ID number of the directory this working directory refers to.

wdCatHint	The Macintosh file system stores a hint so it can quickly find the directory the working directory refers to. Your foreign file system can use this field for its own purposes.
wdProcID	The process ID number passed to the OpenWD request.

The File System Utility routine UTResolveWDCB lets you search the WDCB array. UTResolveWDCB lets you find a WDCB by working directory reference number, or lets you find, one at a time, all WDCBs or all of the WDCBs with a particular process ID.

Note: If your foreign file system attempts to allocate a WDCB in response to an _OpenWD call and UTAllocateWDCB cannot find a free WDCB in the WDCB array, you should return the tmfoErr result from UTAllocateWDCB as your result. The Macintosh file system will then attempt to enlarge the WDCB array and if successful, will retry the _OpenWD call. Because enlarging the WDCB array will change the array's location in memory, you should never save the location of WDCBRecs between calls to your foreign file system. A WDCBRecPtr can only be used within the context of one file system operation.

Drive Queue Elements

The File Manager maintains a list of all disk drives connected to the computer. It maintains this list in the drive queue, which is a standard Operating System queue. The drive queue is initially created at system startup time. Elements are added to the queue at system startup time or when the AddDrive function is called. The drive queue can support any number of drives, limited only by memory space. Each element in the drive queue contains information about the corresponding drive; the structure of a drive queue element is defined by the DrvQEl data type.

Given a drive number, the UTFindDrive File System Utility routine will return a pointer to the associated drive queue element if it is found in the drive queue.

```
struct DrvQEl {
     QElemPtr         qLink;          /* Next queue entry */
     short            qType;          /* Flag for dQDrvSz and
                                         dQDrvSz2 */
     short            dQDrive;        /* Drive number */
     short            dQRefNum;       /* Driver reference number */
     short            dQFSID;         /* File system ID */
     unsigned short   dQDrvSz;        /* Number of logical blocks on
                                         drive */
     unsigned short   dQDrvSz2;       /* Additional field for large
                                         drives */
     };
     typedef struct DrvQEl DrvQEl;
     typedef DrvQEl *DrvQElPtr;
```

Field descriptions

qLink	A pointer to the next entry in the drive queue. This field is initialized when the drive is added to the drive queue by _AddDrive.
qType	The queue type. Used to specify the size of the drive. If the value of qType is 0, the number of logical blocks on the drive is contained in the dQDrvSz field alone. If the value of qType is 1, both dQDrvSz and dQDrvSz2 are used to store the number of blocks; in that case, dQDrvSz2 contains the high-order word of this number and dQDrvSz contains the low-order word.
dQDrive	The drive number of the drive. This field is initialized when the drive is added to the drive queue by _AddDrive.
dQRefNum	The driver reference number of the driver controlling the device on which the volume is mounted. This field is initialized when the drive is added to the drive queue by _AddDrive.
dQFSID	An identifier for the file system handling the volume in the drive; it is zero for volumes handled by the File Manager and nonzero for volumes handled by other file systems.
dQDrvSz	The number of logical blocks on the drive.
dQDrvSz2	An additional field to handle large drives. This field is used only if the qType field contains 1 and then, contains the high-order word specifying the number of logical blocks on the drive.

▲ **Warning:** If the device driver whose reference number is dQRefNum supports the Return Format List _Status call (csCode = 6), the volSize field in the returned FormatListRec with the diCIFmtFlagsCurrentBit set should be used to determine the number of blocks instead of the dQDrvSz and dQDrvSz2 fields in the DrvQEl. Drivers that support Return Format List (for example, the .Sony driver) may not support the dQDrvSz and dQDrvSz2 fields or may use them for other purposes. ▲

There are also four flag bytes preceding each drive queue element. The four flag bytes must be allocated by the disk driver at the same time the drive queue element is allocated. These bytes contain the following information:

Byte	Contents
0	Bit 7=1 if the volume on the drive is locked
1	0 if no disk in drive; 1 or 2 if disk in drive; 8 if nonejectable disk in drive; $FC–$FF if disk was ejected within last 1.5 seconds; $48 if disk in drive is nonejectable but driver wants a call
2	Used internally during system startup
3	Bit 7=0 if disk is single-sided

You can read these flags by subtracting 4 from the beginning of a drive queue element.

After successfully mounting a volume, a foreign file system should change the dqFSID field in the drive queue element to the foreign file system's file system ID number. When the volume is unmounted and the VCB is disposed of with UTDisposeVCB, UTDisposeVCB will clear the dqFSID field in the drive queue element if the device is ejectable.

The dqFSID field in drive queue elements for the Macintosh floppy disk drives are always initially zero. Other disk drivers should initialize the dqFSID field to either the file system ID number of the file system that formatted the disk, or to fsmGenericFSID ($ffff) if the file system cannot be identified.

FILE SYSTEM LOCATION INFORMATION

A file system must be able to determine the file, directory, or volume using information passed to it in a parameter block. This section tells how to use File System Utility routines to make that determination. The section titled "HFS Specifications" in Inside Macintosh: Files lists the various ways files or directories can be specified.

About Volume Reference Numbers

Each volume is given a unique volume reference number when the volume is mounted and the volume's VCB is added to the VCB queue by UTAddNewVCB. A volume reference number is only valid for a particular volume while that volume is mounted. If a volume is unmounted and then mounted again, a new volume reference number will be assigned to the volume.

About Directory ID Numbers

A foreign file system must be able to assign a directory ID number to all directories on a volume. Each directory ID must be unique and cannot be used as a file number for the same volume. For directories other than the root directory (which must be fsRtDirID), the directory ID must be fsUsrCNID (16) or greater.

While a volume is mounted, directory IDs must remain the same. Directory IDs must be kept the same while a volume is unmounted. Macintosh programs and the Alias Manager use directory IDs to locate directories on the volume between volume mounts.

About File Numbers

A foreign file system must be able to assign a file number to all files on a volume. A file number is never passed as a parameter to a file system routine. Each file number must be unique and cannot be used as a directory ID number for the same volume. File numbers for files opened by programs using File Manager requests must be fsUsrCNID (16) or greater. File numbers less than fsUsrCNID are reserved for the root directory ID and for files opened by file systems for their own internal use (for example, the catalog and extents overflow files used by the HFS file system). The File Manager's UnmountVol code returns fBsyErr to the caller and will not call your foreign file system if there are open files on the specified volume with file numbers fsUsrCNID or greater. If your foreign file system open files with file numbers less than fsUsrCNID, your foreign file system is responsible for closing those files before unmounting a volume.

While a volume is mounted, file numbers must remain the same. File numbers should, if at all possible, be kept the same while a volume is unmounted.

A foreign file system will use a file number in the file control block (FCB) of an open file. The file number can be used to find a FCB, and is used to identify multiple access paths to the same file by the UTAdjustEOF and UTCheckPermission File System Utility routines.

About File ID References

File ID references are an optional file system feature that allow the Alias Manager and other programs to track files that have been moved or renamed within a volume. If a foreign file system supports file ID references on its volumes, the foreign file system must be able to assign a file ID reference to a file, and later return that file's location and name given only the file ID reference number, even if the file has been moved or renamed.

File ID reference numbers must be kept the same while a volume is unmounted. Macintosh programs and the Alias Manager use file ID reference numbers to locate files on the volume between volume mounts.

Determining the Volume

Before a foreign file system is called, the Macintosh file system and the File System Manager have determined what volume is the target of the call and what file system owns that volume. The volume is identified by the volume control block parameter passed to foreign file system's HFSCIProc. UTDetermineVol can also be used to determine what volume is the target of a call.

Two file system calls are exceptions: _MountVol and _VolumeMount. _MountVol and _VolumeMount both deal with unmounted volumes (or potentially unmounted volumes), so the volume control block parameter passed to foreign file system's HFSCIProc is unused.

Determining the File or Directory Location

A file's location can be given in a file system call by full pathname, or by partial pathname. A directory's location can be given by full pathname, partial pathname, or directory ID. A foreign file system must determine the directory ID the pathname starts in before the pathname to a file or directory can be parsed. Listing 3-2, GetCurrentDir, shows how to use the contents of the call's parameter block and File System Utility routines to determine the directory parsing should begin in. The dirID input parameter contains the directory ID from the parameter block if that field is valid for the request recieved. Not all requests include a directory ID specification. For example, the old file system calls like PBOpen do not include a directory ID specification. GetCurrentDir returns a pointer to the volume's VCB in volCtrlBlockPtr, the directory where parsing should begin is returned in currentDirectory, and the method used to determine the volume is returned in status.

Listing 3-2. Determining where directory parsing should start

```
OSErr GetCurrentDir(ParmBlkPtr pb, long dirID, VCBPtr *volCtrlBlockPtr,
                    long *currentDirectory, short *status)
{
   OSErr       result;
   short       moreMatches;
```

```
short       vRefNum;
WDCBRecPtr  wdCtrlBlockPtr;
WDPBRec     wdpb;

result = UTDetermineVol(pb, status, &moreMatches, &vRefNum,
                        volCtrlBlockPtr);
if (result == noErr)
{
   switch (*status)
   {
      /* Determined by full pathname */
      case dtmvFullPathame:
         /* Current directory is always the root directory */
         *currentDirectory = fsRtDirID;
         break;

      /* Determined by volume refNum or by drive number */
      case dtmvVRefNum:
      case dtmvDriveNum:
         /* Current directory is dirID if specified; */
         /* otherwise, it's the root directory */
         if (dirID != 0)
            *currentDirectory = dirID;
         else
            *currentDirectory = fsRtDirID;
         break;

      /* Determined by working directory refNum */
      case dtmvWDRefNum:
         /* Current directory is dirID if specified; */
         /* otherwise, it's wdDirID in the WDCB */
         if (dirID != 0)
            *currentDirectory = dirID;
         else
         {
            result = UTResolveWDCB(0, 0, pb->fileParam.ioVRefNum,
                                   &wdCtrlBlockPtr);
            if (result == noErr)
               *currentDirectory = wdCtrlBlockPtr->wdDirID;
         }
         break;

      /* Determined by default volume */
      case dtmvDefault:
         /* Current directory is dirID if specified; */
         /* otherwise, it's the default directory */
         if (dirID != 0)
            *currentDirectory = dirID;
         else
         {
            wdpb.ioNamePtr = NULL; /* the name not needed */
            result = UTGetDefaultVol(&wdpb);
            if (result == noErr)
```

```
                *currentDirectory = wdpb.ioWDDirID;
            }
            break;

         /* This should never happen, but... */
         default:
            result = paramErr;
            break;
      }
   }
   return (result);
}
```

Once the directory ID where parsing will begin has been determined, the pathname (if any) must be parsed to find the object. The process of parsing a pathname consists of getting the next component in a pathname and determining where that component is on the volume, and then repeating those two steps until the end of the pathname is reached. Parsing can go either up or down in a volume's directory heirarchy so a foreign file system must be able to find an object by name within a specified directory to move down in the heirarchy and must be able to find an object's parent directory to move up in the heirarchy. Listing 3-3 shows how to parse a pathname. How a foreign file system finds an object on a volume is specific to the foreign file system.

Listing 3-3. Parsing a pathname

```
struct ObjectInfoRec {
   long     parentDirID;   /* the directory ID of the parent */
   Str31    localName;     /* the local name within said directory */
   Boolean  exists;        /* TRUE if the object exists */
   Boolean  isDir;         /* TRUE if the object is a directory */
   long     objectNumber;  /* directory ID or file number of object */
};
typedef struct ObjectInfoRec ObjectInfoRec;
typedef ObjectInfoRec *ObjectInfoRecPtr;

OSErr ParsePathName (VCBPtr   theVCB, long currentDirectory,
                     StringPtr namePtr, ObjectInfoRecPtr objectInfo)
{
   OSErr       result;
   ParsePathRec   parseRec;
   Str31       lastComponent;

   /* init parseRec to start parsing at beginning of namePtr */
   parseRec.namePtr = namePtr;
   parseRec.startOffset = 0;
   parseRec.ComponentLength = 0;
   parseRec.moreName = false;
   parseRec.foundDelimiter = false;
```

```
/* Get the length of the volume name (if any) and use it to adjust
** the startOffset to point to the beginning of the partial pathname.
** UTParsePathname also ensures the pathname is not NULL or an empty
** string.
*/
result = UTParsePathname(&parseRec.startOffset, parseRec.namePtr);
if (result == noErr)
{
   /* At this point, startOffset is pointing at the first
   ** character of a partial pathname (if any). For example:
   **      namePtr = 'VolName:a:b:c'   (full pathname)
   **  startOffset =         ^
   **      namePtr = ':a:b:c'          (partial pathname)
   **  startOffset =  ^
   **      namePtr = 'a'               (partial pathname)
   **  startOffset =  ^
   **      namePtr = 'VolName:'        (full pathname)
   **  startOffset =         ^
   */

   /* Check for leading delimiter of the partial pathname */
   result = UTGetPathComponentName(&parseRec);
   if (result == noErr)
   {
      if (parseRec.ComponentLength == 0 && parseRec.foundDelimiter)
      {
         /* Get past initial delimiter */
         ++parseRec.startOffset;
      }

      /* At this point, startOffset is pointing at the first
      ** character of the first component name (if any).
      ** For example:
      **      namePtr = 'VolName:a:b:c'   (full pathname)
      **  startOffset =          ^        (moreName = true,
      **                                   componentLength = 0)
      **      namePtr = ':a:b:c'          (partial pathname)
      **  startOffset =   ^               (moreName = true,
      **                                   componentLength = 0)
      **      namePtr = 'a'               (partial pathname)
      **  startOffset =  ^                (moreName = true,
      **                                   componentLength = 0)
      **      namePtr = 'VolName:'        (full pathname)
      **  startOffset =          ^        (moreName = false,
      **                                   componentLength = 0)
      */

      /* Parse until there is no more pathname to parse. */
      while ((result == noErr) && parseRec.moreName)
      {
         /* Search for the next delimiter from startOffset. */
         result = UTGetPathComponentName(&parseRec);
```

```
        if (result == noErr)
        {
           if (parseRec.ComponentLength == 0)
           {
              /* A delimiter was found immediately following another
              ** delimiter. Set current directory to parent of
              ** current directory. If current directory is
              ** fsRtDirID, then return bdNamErr (that handles the
              ** case of 'Root::' or 'Root:SubDir:::', etc.)
              */
              if (currentDirectory != fsRtDirID)
              {
                 /* Call foreign file system specific code to get
                 ** parent of current directory
                 */
                 result = FFSGetParent(theVCB, currentDirectory,
                                       &currentDirectory);
              }
              else
              {
                 /* Pathname tried to go up from root directory */
                 result = bdNamErr;
              }

              /* startOffset = start of next component (if any). */
              ++parseRec.startOffset;
           }
           else if (parseRec.moreName)
           {
              /* A component was found and it isn't the last
              ** component, so it must be a directory.  Make
              ** lastComponent in current directory the new current
              ** directory.
              */
              lastComponent[0] = parseRec.ComponentLength;
              BlockMove((Ptr)((long)parseRec.namePtr +
                    parseRec.startOffset + 1),
                    &lastComponent[1],
                    parseRec.ComponentLength);
              /* Call foreign file system specific code to get
              ** directory by name in currentDirectory
              */
              result = FFSGetDir(theVCB, lastComponent,
                             currentDirectory, &currentDirectory);

              /* startOffset = start of next component (if any). */
              parseRec.startOffset += parseRec.ComponentLength + 1;
           }
        }
     }
```

```
      if (result == noErr)
      {
         /* There is no more pathname to parse. */
         if (parseRec.ComponentLength == 0)
         {
            /* The pathname ended with '::' or the pathname was
            ** simply a volume name ('VolName:'), so current
            ** directory is the object.
            */
            objectInfo->isDir = true;
            objectInfo->objectNumber = currentDirectory;
            /* Call foreign file system specific code to get
            ** information on this directory
            */
            result = FFSGetDirInfo(theVCB, objectInfo);
         }
         else
         {
            /* The pathname ended with 'name:' or 'name', so name is
            ** the object.
            */
            objectInfo->localName[0] = parseRec.ComponentLength;
            BlockMove((Ptr)((long)parseRec.namePtr +
                  parseRec.startOffset + 1),
               &(objectInfo->localName[1]),
               parseRec.ComponentLength);

            objectInfo->parentDirID = currentDirectory;
            /* Call foreign file system specific code to get
            ** information on this file
            */
            result = FFSGetFile(theVCB, objectInfo);
         }
      }
   }
}
else
{
   result = bdNamErr;   /* Default to bad name (if ioNamePtr != NULL
                        ** or ioNamePtr doesn't point to an empty
                        ** string). Otherwise, the code that follows
                        ** will set up the real result.
                        */
   if ((namePtr == NULL) || (namePtr[0] == 0))
   {
      /* pathname was NULL or a zero length string, so we found a
      ** directory at the end of the string
      */
      objectInfo->isDir = true;
      objectInfo->objectNumber = currentDirectory;
      /* Call foreign file system specific code to get information on
      ** this directory
      */
```

```
            result = FFSGetDirInfo(theVCB, objectInfo);
         }
      }
      return (result);
}
```

THE MACINTOSH VOLUME CACHE

The Macintosh operating system uses a volume (disk) cache for input and output operations. The following sections provide both general information about caching and specific information about the volume cache used by the Macintosh operating system.

About the Macintosh Volume Cache

The Macintosh operating system allows users to set aside memory for a general-purpose volume cache. A volume cache is an area in memory that is used to store blocks that have been recently read from or written to a volume. By keeping blocks in memory that are likely to be read or written again, a file system can avoid repeating I/O operations to the volume's device and, as a result, can save a great deal of time. Use of the cache can greatly increase the speed of many applications. The one disadvantage is that memory set aside for the cache is unavailable to applications.

How Users Control the Cache

The Memory control panel allows users to change the amount of memory set aside for the cache. The Memory control panel allows users to reclaim some, but not all, of the memory used by the cache when necessary.

Although a foreign file system cannot determine the size of the cache, there is always at least 32K set aside for the cache.

The Cache and Foreign File Systems

A foreign file system can gain access to the cache by using the cache routines provided by the File System Manager. Foreign file system's should try to use these routines for all input/output operations. There are a number of reasons why a foreign file system should use the cache routines instead of calling a device driver directly:

- Increased throughput: By reducing the number of input/output operations, use of the cache can dramatically increase throughput.
- Users can control the cache: The Control Panel lets users control the amount of memory set aside for caching.
- Easier implementation: The foreign file system does not have to allocate a buffer for an entire block when only part of a block is to be transferred.
- Support for both logical and physical blocks: The Macintosh cache routines can perform operations involving physical blocks on a volume or logical blocks for a file.

- Asynchronous execution: All cache routines that transfer data to or from a volume execute asynchronously. This means that control can be returned from the Macintosh file system to the caller before an input or output operation is complete. (Although to the foreign file system it appears that the cache routine waits until the operation has completed, the Macintosh file system returns control to the caller—unless the original trap was requested to be run synchronously—while waiting for the I/O operation to finish. When the I/O operation is complete, the Macintosh file system returns control to the foreign file system in such a way that normal completion of the foreign file system routine will allow execution of the application's code to resume where it was interrupted.)

▲ **Warning:** It is important to note the use of the cache routines is the only way a foreign file system can perform asynchronous I/O. If a foreign file system calls a device driver directly, all I/O operations are executed synchronously, even if the calling application has requested asynchronous I/O. If your foreign file system is called at interrupt time and the device driver (or another device driver called by the device driver) is busy, calling the device driver directly will deadlock the Macintosh system. For this reason, all calls to most device drivers should be made through the cache routines. ▲

Cache Blocks

Each cache block is made up of header information that identifies the volume and file (if any) from which the data was taken and an offset that identifies how far, in bytes, the block was from the beginning of the volume (for volume I/O) or the beginning of the file (for file I/O) and 512 bytes of data. The structure of cache block headers and where they reside in memory is undefined. Cache routines that work with individual cache blocks always use a pointer to the cache block's data.

The Macintosh volume cache has four categories of cache blocks:

- **Reserved** cache blocks - A cache block that is in use is reserved. A reserved cache block corresponds to a logical block of a file or a physical block of a volume and is not available for reuse. A cache block is reserved with the UTGetBlock function. A reserved cache block can contain clean or dirty data (or no valid data at all if the cache block was obtained for use as a write buffer). Cache blocks should not be reserved between calls to a file system.

- **Released** cache blocks - A cache block that is very likley to be needed again but is not reserved should be released. A released cache block still corresponds to a block of the file or volume, but is available for reuse. A released cache block can contain clean or dirty data.

- **Free** cache blocks - A cache block that isn't likley to be used again should be marked free. A free cache block still contains valid data and is still associated with the corresponding block of the volume, but not with a file on the volume. Dirty cache blocks should be flushed before marking them free. Free cache blocks will be reused before released cache blocks.

- **Trashed** cache blocks - A cache block that will never be used again should be trashed. Trashing a cache block marks the cache block empty. Dirty cache blocks should be flushed before trashing them. Trashed cache blocks will be reused before released or free cache blocks.

As mentioned above, a cache block can be dirty or clean. A cache block is **dirty** when its contents do not match the contents of the corresponding block on the volume. To update the volume, a dirty cache block must be **flushed** by writing its contents out to the volume. This occurs in the following cases:

- The Macintosh operating system replaces a released cache block that is dirty with a block from the volume that is being read into the cache. When this occurs, the dirty cache block is flushed before it is replaced.

- The foreign file system flushes the dirty cache block in response to a _FlushFile, _Close, or _FlushVol call.

- The foreign file system flushes a cache block at some other time (for example, when handling a _Write call).

When using the cache routines, remember that any cache operation may force a previously cached block to be removed from memory. If there are no trashed or free cache blocks, a new block that is read into the cache from the volume replaces the contents of the least recently used of the released blocks (if any) in the cache.

Logical to Physical Block Mapping

The Macintosh cache routines can read or write either logical blocks of a file or physical blocks of a volume. However, because disk driver device calls always read or write physical blocks of a volume, a foreign file system must provide a routine to map logical file blocks to physical volume blocks called the logical to physical routine. The address of the logical to physical routine is passed to the File System Manager in the foreign file system's HFSCIRec and as a parameter to some cache routines. The interface for the logical to physical routine a FSM-based foreign file system must supply is defined in "The HFS Component Interface" chapter.

It is the responsibility of the foreign file system to return an eofErr when a request for data is beyond the current logical end of file. It is the responsibility of the logical to physical routine to return an eofErr error when a cache routine asks for block mapping beyond the current physical end of file.

FILE SYSTEM UTILITY ROUTINES

UTAllocateFCB

Use the UTAllocateFCB function to allocate a FCBRec in the FCB array when opening a file.

```
pascal OSErr UTAllocateFCB(short *fileRefNum,
                           FCBRecPtr *fileCtrlBlockPtr);
```

fileRefNum	Contains a pointer to a short. UTAllocateFCB places the file reference number of the allocated FCBRec into the field referred to by this parameter.
fileCtrlBlockPtr	Contains a pointer to a FCBRecPtr. UTAllocateFCB places a pointer to the allocated FCBRec into the field referred to by this parameter.

If an FCBRec can be allocated in the FCB array, the FCB will be marked in use (fcbFlNm is set to -1), the file reference number will be returned in fileRefNum, and a pointer to the allocated FCBRec will be returned in fileCtrlBlockPtr. The allocated FCBRec is not initialized in any way. It is up to your foreign file system to initialize all fields in the FCBRec.

If an FCB cannot be allocated because there are no FCBRecs left in the FCB array, this function will return a tmfoErr.

Note: If your foreign file system returns the tmfoErr result from UTAllocateFCB as the result to _Open, the Macintosh file system will attempt to enlarge the FCB array and if successful, will retry the _Open call. Because enlarging the FCB array will change the array's location in memory, you should never save the location of a FCBRec between calls to your foreign file system. A FCBRecPtr can only be used within the context of one file system operation.

When you no longer need an allocated FCBRec (either because the file could not be opened or when the file is closed), release it with UTReleaseFCB.

Result codes

noErr	0	No error
tmfoErr	-42	There are no free FCBRecs in the FCB array

UTReleaseFCB

Use the UTReleaseFCB function to release an allocated FCBRec in the FCB array when closing a file.

```
pascal OSErr UTReleaseFCB(short fileRefNum);
```

fileRefNum	Contains the file reference number that specifies the FCBRec to release.

When you no longer need an allocated FCBRec (either because the file could not be opened or when the file is closed), release it with UTReleaseFCB. UTReleaseFCB will release an FCBRec in the FCB array and mark it free (the FCBRec is cleared).

Result codes

noErr	0	No error
fnOpnErr	-38	The file is not open
rfNumErr	-51	The file reference number is not valid

UTLocateFCB

Use the UTLocateFCB function to find a FCBRec in the FCB array.

```
pascal OSErr UTLocateFCB(VCBPtr volCtrlBlockPtr, unsigned long fileNum,
            StringPtr namePtr, short *fileRefNum,
            FCBRecPtr *fileCtrlBlockPtr);
```

volCtrlBlockPtr	Contains a VCBPtr that specifies the volume the open file is on.
fileNum	Contains a file number that specifies the FCBRec to search for. If you want to find the FCBRec by file name, set this parameter to zero.
namePtr	Contains a pointer to a string that specifies the file name to search for. This parameter is ignored if fileNum is not zero.
fileRefNum	Contains a pointer to a short. UTLocateFCB places the file reference number of the FCBRec found into the field referred to by this parameter.
fileCtrlBlockPtr	Contains a pointer to a FCBRecPtr. UTLocateFCB places a pointer to the FCBRec found into the field referred to by this parameter.

UTLocateFCB will find a FCBRec in the FCB array that's on the volume specified by volCtrlBlockPtr. If fileNum is not zero, UTLocateFCB will search the FCB array for the first FCBRec that has a matching file number. If fileNum is zero, UTLocateFCB will search the FCB array for the first FCBRec that has a matching file name. UTLocateFCB always starts searching from the beginning of the FCB array and returns the first match it finds. UTLocateFCB returns the file reference number of the matching FCBRec in fileRefNum and a pointer to the matching FCBRec in fileCtrlBlockPtr.

Result codes

noErr	0	No error
fnfErr	-38	A matching FCB was not found

UTLocateNextFCB

Use the UTLocateNextFCB function to continue seaching for a FCBRec in the FCB array.

```
pascal OSErr UTLocateNextFCB(VCBPtr volCtrlBlockPtr,
                             unsigned long fileNum, StringPtr namePtr,
                             short *fileRefNum,
                             FCBRecPtr *fileCtrlBlockPtr);
```

volCtrlBlockPtr — Contains a VCBPtr that specifies the volume the open file is on.

fileNum — Contains a file number that specifies the FCBRec to search for. If you want to find the FCBRec by file name, set this parameter to zero.

namePtr — Contains a pointer to a string that specifies the file name to search for. This parameter is ignored if fileNum is not zero.

fileRefNum — Contains a pointer to a short. On input, the field referred to by this parameter contains the file reference number of the last match found by UTLocateFCB or UTLocateNextFCB. On output, UTLocateNextFCB places the file reference number of the FCBRec found into the field referred to by this parameter.

fileCtrlBlockPtr — Contains a pointer to a FCBRecPtr. UTLocateFCB places a pointer to the FCBRec found into the field referred to by this parameter.

UTLocateNextFCB will find a FCBRec in the FCB array that's on the volume specified by volCtrlBlockPtr. If fileNum is not zero, UTLocateNextFCB will search the FCB array for a FCBRec that has a matching file number. If fileNum is zero, UTLocateNextFCB will search the FCB array for a FCBRec that has a matching file name. UTLocateNextFCB always starts searching from the FCBRec specified by fileRefNum and returns the next match it finds. UTLocateNextFCB returns the file reference number of the matching FCBRec in fileRefNum and a pointer to the matching FCBRec in fileCtrlBlockPtr.

Result codes

noErr	0	No error
fnfErr	-38	A matching FCB was not found

UTIndexFCB

Use the UTIndexFCB function to index through the open FCBs on a volume.

```
pascal OSErr UTIndexFCB(VCBPtr volCtrlBlockPtr, short *fileRefNum,
                        FCBRecPtr *fileCtrlBlockPtr);
```

volCtrlBlockPtr — Contains a VCBPtr that specifies the volume the open file is on.

fileRefNum	Contains a pointer to a short. On input, the field referred to by this parameter contains the file reference number of the last match found by UTIndexFCB or zero to search for the first FCBRec. On output, UTIndexFCB places the file reference number of the FCBRec found into the field referred to by this parameter.
fileCtrlBlockPtr	Contains a pointer to a FCBRecPtr. UTIndexFCB places a pointer to the FCBRec found into the field referred to by this parameter.

UTIndexFCB will find a FCBRec in the FCB array that's on the volume specified by volCtrlBlockPtr. UTIndexFCB always starts searching from the FCBRec specified by fileRefNum and returns the next FCBRec it finds on the specified volume. UTIndexFCB returns the file reference number of the matching FCBRec in fileRefNum and a pointer to the matching FCBRec in fileCtrlBlockPtr.

Result codes

noErr	0	No error
fnfErr	-38	A matching FCB was not found
rfNumErr	-51	The file reference number is not valid

UTResolveFCB

Use the UTResolveFCB function to map a file reference number to its FCBRec.

```
pascal OSErr UTResolveFCB(short fileRefNum,
                          FCBRecPtr *fileCtrlBlockPtr);
```

fileRefNum	Contains a file reference number that specifies the FCBRec.
fileCtrlBlockPtr	Contains a pointer to a FCBRecPtr. UTResolveFCB places a pointer to the FCBRec found into the field referred to by this parameter.

Given a file reference number, UTResolveFCB returns a pointer to a FCBRec in fileCtrlBlockPtr.

If the file reference number is valid, UTResolveFCB will return noErr; even if the file is not open. You can use UTCheckFileRefNum to see if a file reference number refers to an open file.

Result codes

noErr	0	No error
rfNumErr	-51	The file reference number is not valid

UTAllocateVCB

Use the UTAllocateVCB function to allocate memory in the system heap for a new VCB.

```
pascal OSErr UTAllocateVCB(unsigned short *sysVCBLength,
                           VCBPtr *volCtrlBlockPtr,
                           unsigned short addSize);
```

sysVCBLength	Contains a pointer to an unsigned short. UTAllocateVCB places the size of the system VCB into the field referred to by this parameter.
volCtrlBlockPtr	Contains a pointer to an VCBPtr. UTAllocateVCB places a pointer to the new VCB into the field referred to by this parameter.
addSize	Contains the amount of addition storage to allocate for the foreign file system's private use, or zero.

UTAllocateVCB will allocate cleared memory from the system heap for a new VCB. The amount of memory allocated by UTAllocateVCB is determined by adding the size of the system VCB structure to the amount of additional memory needed for the foreign file system's private use. A pointer to the new VCB is returned in volCtrlBlockPtr and the size of the system VCB structure is returned in sysVCBLength. Thus, the additional foreign file system private data (if any) starts at volCtrlBlockPtr + sysVCBLength.

Result codes

noErr	0	No error
memFullErr	-108	The VCB could not be allocated; no memory available

UTAddNewVCB

Use the UTAddNewVCB function to add a new VCB to the VCB queue and assign a volume reference number to the volume.

```
pascal OSErr UTAddNewVCB(short driveNum, short *vRefNum,
                         VCBPtr volCtrlBlockPtr);
```

driveNum	Contains the drive number of the drive the volume is on.
vRefNum	Contains a pointer to a short. UTAddNewVCB places the volume reference number assigned to the volume into the field referred to by this parameter.
volCtrlBlockPtr	Contains a pointer to the VCB to add to the VCB queue.

UTAddNewVCB assigns an unused volume reference number to a VCB and adds it to the VCB queue. The volume reference number is returned in vRefNum.

The following fields in the VCB are also initialized by UTAddNewVCB

- qType is assigned fsQType (5).
- vcbDrvNum is assigned the drive number of the drive the volume is on
- vcbDRefNum is assigned the driver reference number of the drive the volume is on

- vcbVRefNum is assigned the volume reference number
- vcbBufAdr may be assigned a value for the system's cache use.

If the VCB is the first VCB added to the VCB queue (this will probably never happen for a foreign file system), UTAddNewVCB will also make the volume the default volume.

Result codes

noErr	0	No error
paramErr	-50	volCtrlBlockPtr was NULL
nsDrvErr	-56	The drive specified by driveNum could not be found in the drive queue

UTDisposeVCB

Use the UTDisposeVCB function to dispose of a VCB allocated with the UTAllocateVCB function.

```
pascal OSErr UTDisposeVCB(VCBPtr volCtrlBlockPtr);
```

volCtrlBlockPtr Contains a pointer to the VCB to dispose.

UTDisposeVCB performs the following

- Close all working directories on the volume
- Remove the VCB from the VCB queue
- If the volume's drive is ejectable, clear the dQFSID field in the DrvQEl of the volume's drive (a disk owned by another file system could be the next disk inserted into the ejectable drive)
- If the volume is the default volume, clear the system's default volume
- Dispose of the memory used by the VCB

You should always use UTDisposeVCB to dispose of your VCB instead of freeing the memory yourself.

Result codes

noErr	0	No error
nsDrvErr	-56	The volume's drive could not be found in the drive queue

UTLocateVCBByRefNum

Use the UTLocateVCBByRefNum function to locate a VCB using a volume reference number, drive number, or working.directory number.

```
pascal OSErr UTLocateVCBByRefNum(short refNum, short *vRefNum,
                                 VCBPtr *volCtrlBlockPtr);
```

refNum — Contains a volume reference number, drive number, or working directory number.

vRefNum — Contains a pointer to a short. UTLocateVCBByRefNum places the reference number of the volume found into the field referred to by this parameter.

volCtrlBlockPtr — Contains a pointer to a VCBPtr. UTLocateVCBByRefNum places a pointer to the VCB found into the field referred to by this parameter.

UTLocateVCBByRefNum finds the VCB associated with a volume reference number, drive number, or working.directory number and returns a pointer to the VCB in volCtrlBlockPtr and the volume reference number in vRefNum.

Result codes

noErr	0	No error
rfNumErr	-51	The working directory reference number is invalid
nsvErr	-56	A matching volume could not be found in the VCB queue

UTLocateVCBByName

Use the UTLocateVCBByName function to locate the first VCB that matches the volume name specified.

```
pascal OSErr UTLocateVCBByName(StringPtr namePtr, short *moreMatches,
                               short *vRefNum, VCBPtr *volCtrlBlockPtr)
```

namePtr — Contains a pointer to a volume name (which must end with a colon ':' character) or to a full pathname.

moreMatches	Contains a pointer to a short. If there is another volume with a matching name, UTLocateVCBByName places the length of the volume name (not including the trailing colon) into the field referred to by this parameter. If this is the last volume with a matching name or no matches are found, UTLocateVCBByName places zero into the field referred to by this parameter.
vRefNum	Contains a pointer to a short. UTLocateVCBByName places the reference number of the volume found into the field referred to by this parameter.
volCtrlBlockPtr	Contains a pointer to a VCBPtr. UTLocateVCBByName places a pointer to the VCB found into the field referred to by this parameter.

UTLocateVCBByName finds the first VCB with the specified volume name and returns a pointer to the VCB in volCtrlBlockPtr and the volume reference number in vRefNum.

UTLocateVCBByName also checks to see if there are more volumes in the VCB queue with the specified name. If there are, moreMatches returns the length of the volume name. If the volume found is the only volume with the specified name (or if no matching volumes are found), moreMatches returns zero. To find the next volume with a matching name, use the UTLocateNextVCB routine.

Result codes

noErr	0	No error
bdNamErr	-37	namePtr was NULL, or namePtr does not point to a volume name or full pathname
nsvErr	-56	A matching volume could not be found in the VCB queue

UTLocateNextVCB

Use the UTLocateNextVCB function to continue seaching for a VCB by name in the VCB queue.

```
pascal OSErr UTLocateNextVCB(StringPtr namePtr, short *moreMatches,
                             short *vRefNum, VCBPtr *volCtrlBlockPtr);
```

namePtr	Contains a pointer to a volume name (which must end with a colon ‘:’ character) or to a full pathname.
moreMatches	Contains a pointer to a short. On input, the field referred to by this parameter contains the length of the volume name as returned by a previous call to UTLocateVCBByName or UTLocateNextVCB, or zero to force UTLocateNextVCB to re-parse the pathname. On output, UTLocateNextVCB places the length of the volume name (not including the trailing colon) into the field referred to by this parameter. If this is the last volume with a matching name or no matches are found UTLocateNextVCB places zero into the field referred to by this parameter.

vRefNum	Contains a pointer to a short. UTLocateNextVCB places the reference number of the volume found into the field referred to by this parameter.
volCtrlBlockPtr	Contains a pointer to a VCBPtr. UTLocateNextVCB places a pointer to the VCB found into the field referred to by this parameter.

UTLocateNextVCB finds the next VCB with the specified volume name. UTLocateNextVCB always starts searching from the VCB specified by volCtrlBlockPtr and returns the next VCB it finds in the VCB queue. UTLocateNextVCB returns a pointer to the VCB in volCtrlBlockPtr and the volume reference number in vRefNum.

After finding a match, UTLocateNextVCB also checks to see if there are more volumes in the VCB queue with the specified name. If there are, moreMatches returns the length of the volume name. If the volume found is the only volume with the specified name (or if no matching volumes are found), moreMatches returns zero.

Result codes

noErr	0	No error
bdNamErr	-37	namePtr was NULL, or namePtr does not point to a volume name or full pathname
nsvErr	-56	A matching volume could not be found in the VCB queue

UTAllocateWDCB

Use the UTAllocateWDCB function to allocate a working directory in the WDCB array.

```
pascal OSErr UTAllocateWDCB(WDPBPtr paramBlock);
```

paramBlock Contains a pointer to a working directory parameter block.

Parameter Block

↔	ioVRefNum	short	On input, contains the volume reference number. On output, contains the working directory reference number.
←	ioWDCreated	short	If a working directory is not created, zero is returned. If a working directory is created, non-zero is returned.
→	ioWDProcID	long	Contains the working directory user identifier.
→	ioWDDirID	long	Contains the working directory's directory ID.

UTAllocateWDCB uses the ioVRefNum, ioWDProcID, and ioPDDirID fields in the WDPBRec to allocate a working directory in the WDCB array. UTAllocateWDCB places the working directory reference number into the ioVRefNum field of the WDPBRec.

If the specified directory has already been made a working directory using the same ioWDProcID value, a new working directory will not be allocated; instead, the existing working directory reference number will be returned and ioWDCreated will be set to zero.

Note: The ioWDCreated field is named filler1 in current versions of Files.C.

If a WDCB array cannot be allocated, this function will return a tmfoErr.

Note: If all WDCBRecs in the WDCB array are in use, an attempt will be made to enlarge the WDCB array. Because enlarging the WDCB array will change the array's location in memory, you should never save the location of a WDCBRec between calls to your foreign file system. A WDCBRecPtr can only be used within the context of one file system operation.

When you no longer need an allocated working directory, release it with UTReleaseWDCB.

Result codes

noErr	0	No error
tmfoErr	-42	There are no free WDCBRecs in the WDCB array

UTReleaseWDCB

Use the UTReleaseWDCB function to release an allocated working directory in the WDCB array when closing a working directory.

```
pascal OSErr UTReleaseWDCB(short wdRefNum);
```

wdRefNum — Contains the working directory reference number that specifies the working directory to release.

When you no longer need an allocated working directory, release it with UTReleaseWDCB. UTReleaseWDCB will release an WDCBRec in the WDCB array and mark it free.

Result codes

noErr	0	No error
rfNumErr	-51	The working directory reference number is not valid or does not refer to an open working directory

UTResolveWDCB

Use the UTResolveWDCB function to find a WDCBRec in the WDCB array.

```
pascal OSErr UTResolveWDCB(long procID, short wdIndex, short wdRefNum,
                           WDCBRecPtr *wdCtrlBlockPtr);
```

procID	Contains a working directory user identifier or zero
wdIndex	Contains an index.
wdRefNum	Contains the working directory reference number that specifies the working directory, or contains a volume reference number or drive number that specifies the volume.
wdCtrlBlockPtr	Contains a pointer to a WDCBRecPtr. UTResolveWDCB places a pointer to the WDCBRec found into the field referred to by this parameter.

UTResolveWDCB locates a WDCBRec in the WDCB array according to these rules:

- If wdIndex is zero or negative, UTResolveWDCB uses the working directory reference number in wdRefNum to find the WDCBRec. The procID parameter is ignored in this case.
- If wdIndex is positive and procID is zero, UTResolveWDCB finds the WDCBRec whose working directory index is wdIndex on the volume specified by wdRefNum.
- If wdIndex is positive and procID is non-zero, UTResolveWDCB finds the WDCBRec whose working directory index is wdIndex with the working directory user identifier specified by procID on the volume specified by wdRefNum.

Result codes

noErr	0	No error
nsvErr	-35	No matching working directory was found by the indexed search
rfNumErr	-51	The working directory reference number is not valid or does not refer to an open working directory

UTFindDrive

Use the UTFindDrive function to find a drive queue element in the drive queue.

```
pascal OSErr UTFindDrive(short driveNum, DrvQElPtr *driveQElementPtr);
```

driveNum	Contains the drive number that specifies the drive.
driveQElementPtr	Contains a pointer to a DrvQElPtr. UTFindDrive places a pointer to the drive queue element found into the field referred to by this parameter.

UTFindDrive locates a drive queue element in the drive queue. If a matching drive queue element is found and its dQFSID field is zero, UTFindDrive will return noErr. If a matching drive queue element is found and its dQFSID field is not zero (when the drive is owned by a foreign file system), UTFindDrive will return extFSErr.

Result codes

noErr	0	A matching drive queue element was found with a zero dQFSID field
nsDrvErr	-56	No matching drive queue element was found
extFSErr	-58	A matching drive queue element was found with a non-zero dQFSID field

UTAdjustEOF

Use the UTAdjustEOF function to adjust the logical and physical EOF of all open FCBs for the specified file fork.

```
pascal OSErr UTAdjustEOF(short fileRefNum);
```

fileRefNum Contains a file reference number that specifies the reference FCBRec.

UTAdjustEOF finds the open FCB specified by fileRefNum and gets that file's file number and VCBPtr. UTAdjustEOF then copies the logical end of file (fcbEOF) and physical end of file (fcbPLen) from the specified FCBRec to all other open FCBRecs that refer to the same file fork.

Result codes

noErr	0	No error
rfNumErr	-51	The file reference number is not valid

UTSetDefaultVol

Use UTSetDefaultVol to set the default volume and working directory in response to _SetVol or _HSetVol.

```
pascal OSErr UTSetDefaultVol(long nodeHint, unsigned long dirID,
                             short refNum);
```

nodeHint Contains a long that specifies optional file system specific information about the directory.

dirID Contains a directory ID or 0.

refNum Contains a volume reference number or a working directory number.

The parameters you pass to UTSetDefaultVol depend on which file system call you are responding to, _SetVol or _HSetVol, and on the parameters passed the file system call.

If you are responding to _SetVol and ioVRefNum is a working directory number, then refNum must be set to the working directory number (ioVRefNum), dirID must be 0, and nodeHint is ignored (nodeHint information is obtained from the working directory specified).

If you are responding to _SetVol and ioVRefNum is a volume reference number or drive number, then refNum must be the volume reference number (obtained from vcbVRefNum in the VCB), dirID must be set to fsRtDirID, and nodeHint can contain optional file system specific information about the directory that you supply.

If you are responding to _HSetVol, then refNum must be the volume reference number (obtained from vcbVRefNum in the VCB), dirID must be be the directory specified by the combination of ioVRefNum, ioDirID and ioNamePtr, and nodeHint can contain optional file system specific information about the directory that you supply.

Result codes

noErr	0	No error
rfNumErr	-51	The refNum is invalid

UTGetDefaultVol

Use UTGetDefaultVol to get the default volume and working directory in response to _GetVol or _HGetVol.

```
pascal OSErr UTGetDefaultVol(WDPBPtr paramBlock);
```

paramBlock Contains a pointer to a working directory parameter block.

Parameter Block

←	ioNamePtr	long	Contains a pointer to a Str31 or NULL. If not NULL, UTGetDefaultVol will return the volume's name in the Str31 specified by this field.
←	ioVRefNum	short	UTGetDefaultVol places the working directory number of the default volume in this field.
←	ioWDProcID	long	UTGetDefaultVol places the working directory user identifier in this field.
←	ioWDVRefNum	short	UTGetDefaultVol places the volume reference number of the default volume in this field.
←	ioWDDirID	long	UTGetDefaultVol places the directory ID number of the default directory in this field.

UTGetDefaultVol returns the default volume name, the default volume's working directory number, the default volume's user identifier, the default volume's volume reference number, and the default directory's directory ID number.

Result codes

noErr	0	No error
nsvErr	-35	The default directory is not set

UTEjectVol

Use UTEjectVol to eject a volume and mark it offline.

```
pascal OSErr UTEjectVol(VCBPtr volCtrlBlockPtr);
```

volCtrlBlockPtr Contains a pointer to the VCB of the volume to eject and mark offline.

UTEjectVol ejects and marks offline the volume specified by volCtrlBlockPtr. If the volume has already been ejected, UTEjectVol does nothing and returns noErr. If the volume is already offline, but has not been ejected, UTEjectVol will eject it.

If the volume's drive is ejectable, UTEjectVol clears the dQFSID field in the DrvQEl of the volume's drive (a disk owned by another file system could be the next disk inserted into the ejectable drive).

Note: Before calling UTEjectVol, the foreign file system must flush any cached data associated with the volume using UTFlushCache. If the device is ejectable, UTFlushCache's fcOption should be set to fcTrashMask. If the device is not ejectable, UTFlushCache's rbOption should be set to rbDefault.

Result codes

noErr	0	No error
paramErr	-50	volCtrlBlockPtr does not point to a valid VCB
any of the result codes returned by the disk driver to an eject call		

UTCheckWDRefNum

Use UTCheckWDRefNum to make sure a working directory reference number is valid and open.

```
pascal OSErr UTCheckWDRefNum(short wdRefNum);
```

wdRefNum Contains the working directory reference number.

UTCheckWDRefNum will return noErr if a working directory reference number is valid and open.

Result codes

noErr	0	No error
rfNumErr	-51	The working directory reference number is not valid or does not refer to an open working directory

UTCheckFileRefNum

Use UTCheckFileRefNum to make sure a file reference number is valid and open.

```
pascal OSErr UTCheckFileRefNum(short fileRefNum);
```

fileRefNum Contains the file reference number.

UTCheckFileRefNum will return noErr if a file reference number is valid and open.

Result codes

noErr	0	No error
rfNumErr	-51	The file reference number is not valid or does not refer to an open file

UTCheckVolRefNum

Use UTCheckVolRefNum to make sure a volume reference number is valid.

```
pascal OSErr UTCheckVolRefNum(short vRefNum);
```

vRefNum Contains the volume reference number.

UTCheckVolRefNum will return noErr if a volume reference number is valid. The vRefNum parameter must be a volume reference number, not a driver number, zero, or a working directory reference number.

Result codes

noErr	0	No error
rfNumErr	-51	The volume reference number is not valid

UTCheckPermission

Use UTCheckPermission to check the file access permission requested in response to _Open or _OpenRF if your foreign file system is using the Macintosh file system's access permission model.

```
pascal OSErr UTCheckPermission(VCBPtr volCtrlBlockPtr, short *modByte,
                                    unsigned long fileNum,
                                    ParmBlkPtr paramBlock);
```

volCtrlBlockPtr — Contains a VCBPtr that specifies the volume the file to be opened is on.

modByte — Contains a pointer to a short. On input, the field referred to by this parameter contains a value that UTCheckPermission uses to check the file access permission. On output, UTCheckPermission places the corrected value for fcbFlags and fcbTypByt into the field referred to by this parameter.

fileNum — Contains a file number that specifies the file to be opened.

paramBlock — Contains a pointer to the parameter block used to make the _Open or _Open request.

UTCheckPermission will check the file access permission requested in response to _Open or _OpenRF to see if it is possible using the Macintosh file system's access permission model.

Before making this call, the foreign file system must determine what file number the open file will have. The file number along with the VCBPtr will allow UTCheckPermission to determine if the requested access permission will conflict with any other access paths to the file.

The foreign file system must also read the file's catalog entry to determine if the file is locked or unlocked. If the file is locked, then the fcbWriteBit and fcbFileLockedBit must be set in the fcbFlags byte (the high byte) of the modByte parameter. The fcbResourceBit must be set in the fcbFlags byte of modByte if responding to an _OpenRF request. All other bits in the fcbFlags byte of modByte must be clear.

The fcbTypByt (the low byte) of modByte was used for the file version by the MFS file system. The fcbTypByt of the modByte must be zero because the Resource Manager, Segment Loader, and Standard File Package won't operate on files with non-zero version numbers.

Table 3-1 lists the valid values that can be used as input values for the modByte parameter.

Table 3-1. Valid values for UTCheckPermission's modByte parameter

Fork to open	Locked state of file	Input value of modByte
data	unlocked	$0000
data	locked	$2100
resource	unlocked	$0200
resource	locked	$2300

If UTCheckPermission returns noErr, then the value returned in modByte contains the corrected values for fcbFlags and fcbTypByt and may be copied into the FCBRec for the new access path.

If UTCheckPermission returns opWrErr, then the file reference number of the existing exclusive-write path is returned in the ioRefNum field of the parameter block.

For a description of the Macintosh file system's access permission model, see Inside Macintosh: Files pages 2-7 and 2-8 and the description of PBHOpen and PBHOpenRF on pages 2-184 through 2-186.

Result codes

noErr	0	No error
opWrErr	-49	fsWrPerm, fsRdWrPerm, or fsRdWrShPerm was requested on a file with an existing exclusive-write access path
permErr	-54	fsWrPerm, fsRdWrPerm, or fsRdWrShPerm was requested on a locked file

UTCheckVolOffline

Use UTCheckVolOffline to make sure a volume reference number is valid and the volume is online.

```
pascal OSErr UTCheckVolOffline(short vRefNum);
```

vRefNum Contains the volume reference number.

UTCheckVolOffline will return noErr if a volume reference number is valid and the volume is online.

Result codes

noErr	0	No error
rfNumErr	-51	The volume reference number is not valid
volOffLinErr	-53	The volume is offline

UTCheckVolModifiable

Use UTCheckVolModifiable to see if a volume can be written to.

```
pascal OSErr UTCheckVolModifiable(short vRefNum);
```

vRefNum Contains the volume reference number.

UTCheckVolModifiable checks the volume's vcbAtrb field in its VCB to determine if the volume is locked by hardware or software.

Result codes

noErr	0	No error
wPrErr	-44	The volume is write protected by hardware
vLckdErr	-46	The volume is locked by software

UTCheckFileModifiable

Use UTCheckFileModifiable to see if a file can be written to.

```
pascal OSErr UTCheckFileModifiable(short fileRefNum);
```

fileRefNum Contains the file reference number.

UTCheckFileModifiable checks the volume's vcbAtrb field in its VCB to determine if the volume is locked by hardware or software and checks the file's fcbFlags in its FCBRec to determine if the file can be written to through the access path specified by fileRefNum.

Result codes

noErr	0	No error
wPrErr	-44	The volume is write protected by hardware
vLckdErr	-46	The volume is locked by software
wrPermErr	-61	The file access path does not have write permission

UTCheckDirBusy

Use UTCheckDirBusy to see if a directory is an open working directory.

```
pascal OSErr UTCheckDirBusy(VCBPtr volCtrlBlockPtr,
                            unsigned long dirID);
```

volCtrlBlockPtr Contains a VCBPtr that specifies the volume the directory is on.

dirID Contains the directory ID.

UTCheckDirBusy checks to see if the directory specified by dirID is an open working directory on the volume specified by volCtrlBlockPtr.

Result codes

noErr	0	No error
fBsyErr	-47	The directory is an open working directory

UTDetermineVol

Use UTDetermineVol to determine what volume a parameter block refers to and how that determination was made.

```
pascal OSErr UTDetermineVol(ParmBlkPtr paramBlock, short *status,
                            short *moreMatches, short *vRefNum,
                            VCBPtr *volCtrlBlockPtr);
```

paramBlock — Contains a pointer to the parameter block to evaluate.

status — Contains a pointer to a short. UTDetermineVol places the decisive factor used to determine what volume paramBlock refers to into the field referred to by this parameter. The status values returned are:

dtmvError	= 0	could not determine volume
dtmvFullPathame	= 1	determined by full pathname
dtmvVRefNum	= 2	determined by volume reference number
dtmvWDRefNum	= 3	determined by working directory reference number
dtmvDriveNum	= 4	determined by drive number
dtmvDefault	= 5	determined by default volume

moreMatches — Contains a pointer to a short. If the volume was determined by full pathname and more than one mounted volume has the volume name passed by paramBlock, UTDetermineVol places a non-zero value (the length of the volume name) into the field referred to by this parameter. The value returned the field referred to by this parameter should be ignored if status is not dtmvFullPathame.

vRefNum — Contains a pointer to a short. UTDetermineVol places the volume reference number into the field referred to by this parameter.

volCtrlBlockPtr — Contains a pointer to an VCBPtr. UTDetermineVol places a pointer to the VCB into the field referred to by this parameter.

UTDetermineVol uses the parameter block pointed to by paramBlock to determine what volume to use and returns the decisive factor used to determine the volume. The decisive factor used to determine the volume along with the other contents of the parameter block are used to find the file or directory the parameter block refers to.

UTDetermineVol determines the volume by searching in the following order:

1. Full pathname - This method is used if ioNamePtr is not NULL and points to a full pathname (see Inside Macintosh: Files for the definition of a full pathname). The first component of the pathname pointed to by the ioNamePtr field in the parameter block is used as the volume name and UTDetermineVol attempts to find the volume by name.

2. Reference number - The ioVRefNum field in the parameter block is used to determine the volume. If ioVRefNum is zero, UTDetermineVol attempts to find the default volume (if the default volume is set up). If ioVRefNum is positive, UTDetermineVol attempts to find the volume by drive number. If ioVRefNum is negative, UTDetermineVol first attempts to find the volume by volume reference number and it that fails, then UTDetermineVol attempts to find the volume by working directory reference number

▲ **Warning:** The PBGetCatInfo and PBGetFInfo File Manager calls do not use the ioNamePtr field in a parameter block as an input when the ioFDirIndex field in a parameter block is non-zero. Before calling UTDetermineVol in these cases, you must save the value of ioNamePtr and set ioNamePtr to NULL so that UTDetermineVol will not attempt to determine the volume by full pathname. After calling UTDetermineVol, you must restore the original value in ioNamePtr. ▲

If the volume cannot be determined, UTDetermineVol will fail with paramErr and the field referred to by the status parameter will be set to dtmvError.

Result codes

noErr	0	No error
paramErr	-50	The parameter block does not refer to a mounted volume

UTParsePathname

Use UTParsePathname to detemine if a pathname is valid, if the pathname is a partial or full pathname, and the volume name length if the pathname is a full pathname.

```
pascal OSErr UTParsePathname(short *volNamelength, StringPtr namePtr);
```

volNamelength — Contains a pointer to a short. If the pathname is a full pathname, UTParsePathname places the length of the volume name into the field referred to by this parameter. If the pathname is a partial pathname, UTParsePathname places zero into the field referred to by this parameter.

namePtr — Contains a pointer to a Pascal string that specifies the pathname.

UTParsePathname determines if namePtr points to a valid pathname, if the pathname is a partial or full pathname, and the volume name length if the pathname is a full pathname. See Inside Macintosh: Files for the definition of a full and partial pathnames. UTParsePathname cannot parse pathnames with more than 31 consecutive colon separators.

Result codes

noErr	0	No error
bdNamErr	-37	namePtr is NULL, namePtr points to a zero length string, the pathname doesn't have correct syntax, or pathname has more than 31 consecutive colon separators

UTGetPathComponentName

Use UTGetPathComponentName to parse the components in a pathname.

```
pascal OSErr UTGetPathComponentName(ParsePathRecPtr parseRec);
```

parseRec Contains a pointer to a ParsePathRec.

parseRec

→	namePtr	StringPtr	Contains a pointer to a Pascal string that contains the pathname.
→	startOffset	short	Contains the offset into the pathname where the parsing will begin. Offset 0 (not 1) is the first character of the pathname.
←	componentLength	short	UTGetPathComponentName places the length of the component into this field.
←	moreName	short	If there is no more pathname left after the parse, UTGetPathComponentName places zero into this field; otherwise, a non-zero value is placed into this field.
←	foundDelimiter	short	If parsing stops because a delimiter (a colon character) is found, UTGetPathComponentName places a non-zero value into this field; otherwise, zero is placed into this field.

UTGetPathComponentName parses a pathname to get the individual components that make up the pathname.

The namePtr field in the ParsePathRec passed to UTGetPathComponentName must point to the pathname to parse. The startOffset field in the ParsePathRec tells UTGetPathComponentName where to start looking for a pathname component.

UTGetPathComponentName returns the length of the component at startOffset in the componentLength field of the ParsePathRec, indicates if the parse found the end of the pathname in the moreName field of the ParsePathRec, and indicates if the parse stopped because a delimiter character was found.

Result codes

noErr	0	No error
bdNamErr	-37	namePtr is NULL, namePtr points to a zero length string, the pathname doesn't have correct syntax, or pathname has more than 31 consecutive colon separators

UTGetBlock

Use UTGetBlock to get a cache block in the Macintosh volume cache.

```
pascal OSErr UTGetBlock(short refNum, void *log2PhyProc,
                        unsigned long blockNum, short gbOption,
                        Ptr *buffer);
```

refNum — Contains the file or volume reference number.

log2PhyProc — Contains a pointer to the logical to physical routine or NULL if refNum is a volume refererence number.

blockNum — Contains the logical block number of the file, or the physical block number of the volume.

gbOption — Contains the get block options. The option values you can pass are:

gbDefault	read from disk if block is not found in the cache
gbRead	always read block from disk (forced read)
gbExist	get block only if it is already in the cache as a released block.
gbNoRead	do not read block if it is not already in the cache
gbRelease	this option can be added to any of the other gbOptions to release the block immediately after getting it

buffer — Contains a pointer to a Ptr. UTGetBlock places the memory address of the cache block into the field referred to by this parameter.

UTGetBlock gets a cache block in the Macintosh volume cache and returns the address of the cache block in the buffer parameter. If refNum is a volume reference number, the blockNum is the physical block on the volume. If refNum is a file reference number, then GetBlock calls the logical to physical routine to map blockNum, the logical file block, to a physical block on the volume.

UTGetBlock lets you specify several options with the gbOption parameter:

- gbDefault should be used to get a disk block from the cache if it's in the cache, or to bring a disk block into the cache if it isn't in the cache.
- gbRead should be used to force a block to be read from from the disk. For example, you'd use the gbRead option when handling a _Read call where the read-verify mode is requested by the caller.
- gbNoRead should be used to get a cache block who's contents will be completely overwritten.
- gbExist should be used for getting a disk block which should already be in the cache as a released block. A free cache blocks is not found if gbExist is specified. If UTGetBlock with the gbExists option fails, use UTBlockInFQHashP to see if a physical block on the volume is in a free cache block in the Macintosh volume cache and if so, retry UTGetBlock with the gbDefault option.
- gbRelease can be added to any of the other gbOptions to release the cache block immediately after getting it. This saves the step of releasing the block with UTReleaseBlock when you know you won't be making another call to the Macintosh Volume Cache (which could reuse the cache block) before the cache block is used.

Important note: UTGetBlock can potentially return an internal positive error result code. Because the file system should never return a positive result, your foreign file system should return ioErr if it cannot recover gracefully from one of the positive error results.

▲ **Warning:** UTGetBlock may only be called from a foreign file system handling a request through its HFSCIProc. Attempts to call UTGetBlock outside of the context of the HFSCIProc will cause a system crash. ▲

Result codes

noErr	0	No error
chNoBuf	1	Cache internal error - no free cache buffers (all reserved)
chInUse	2	Cache internal error - requested block is reserved
chNotfound	3	Cache internal error - requested block was not found (only if gbExist option)
ioErr	-36	Device I/O error
offLinErr	-65	Device was offline
notEnoughMemoryErr	-620	VM was not able to hold down enough physical memory

UTReleaseBlock

Use UTReleaseBlock to release a reserved cache block.

```
pascal OSErr UTReleaseBlock(Ptr buffer, short rbOption);
```

buffer — Contains a pointer to the cache block to release.

rbOption — Contains the release block options. The option values you can pass are:

rbDefault	release the cache block
rbWrite	release, write the cache block to disk (forced write) - this will also mark the cache block clean
rbTrash	mark the cache block trashed
rbDirty	release and mark the cache block dirty (deferred write)
rbFree	mark the cache block free

UTReleaseBlock releases the reserved cache block specified in the buffer parameter. The rbOption parameter also allows UTReleaseBlock to write the cache block to disk, mark the cache block dirty, free the cache block, or trash the cache block.

Important note: UTReleaseBlock can potentially return an internal positive error result code. Because the file system should never return a positive result, your foreign file system should return ioErr if it cannot recover gracefully from the positive error result.

▲ **Warning:** UTReleaseBlock may only be called from a foreign file system handling a request through its HFSCIProc. Attempts to call UTReleaseBlock outside of the context of the HFSCIProc will cause a system crash. ▲

Result codes

noErr	0	No error
chNotInUse	1	Cache internal error - block being released was not in use
ioErr	-36	Device I/O error (only if rbWrite option)
offLinErr	-65	Device was offline (only if rbWrite option)
notEnoughMemoryErr	-620	VM was not able to hold down enough physical memory (only if rbWrite option)

UTFlushCache

Use UTFlushCache to flush all dirty cache blocks associated with a file or volume.

```
pascal OSErr UTFlushCache(short refNum, short fcOption);
```

refNum — Contains the file or volume reference number.

fcOption — Contains the flush block options. The option values you can pass are:

fcDefault	flush the cache blocks
fcTrash	mark the cache blocks trashed after flushing them
fcFree	mark the cache blocks free after flushing them

UTFlushCache flushes all dirty blocks associated with the file or volume specified by refNum. The fcOption parameter also allows UTFlushCache to free the cache blocks, or trash the cache blocks after flushing them.

Note: When the fcTrash or fcFree option is used, all cache blocks associated with the file or volume are trashed or freed - not just those blocks that were dirty and flushed.

▲ **Warning:** UTFlushCache may only be called from a foreign file system handling a request through its HFSCIProc. Attempts to call UTFlushCache outside of the context of the HFSCIProc will cause a system crash. ▲

Result codes

noErr	0	No error

UTMarkDirty

Use UTMarkDirty to mark a cache block dirty.

```
pascal OSErr UTMarkDirty(Ptr buffer);
```

buffer Contains a pointer to the cache block to mark dirty.

UTMarkDirty marks the specified cache block dirty.

Result codes

noErr	0	No error

UTTrashVolBlocks

Use UTTrashVolBlocks to trash all file and volume cache blocks associated with a volume.

```
pascal OSErr UTTrashVolBlocks(VCBPtr volCtrlBlockPtr);
```

volCtrlBlockPtr Contains a VCBPtr that specifies the volume.

UTTrashVolBlocks trashes all file and volume cache blocks associated with the volume specified by volCtrlBlockPtr. That includes all released blocks and all free blocks. This routine must be used when a volume is unmounted or ejected.

Result codes

noErr	0	No error

UTTrashFileBlocks

Use UTTrashFileBlocks to trash all cache blocks associated with a file.

```
pascal OSErr UTTrashFileBlocks(VCBPtr volCtrlBlockPtr,
                               unsigned long fileNum)
```

volCtrlBlockPtr Contains a VCBPtr that specifies the volume the file is on.

fileNum Contains the file’s file number.

UTTrashFileBlocks trashes all cache blocks associated with the file specified by fileNum on the volume specified by volCtrlBlockPtr. This routine is typically used when a file is deleted.

Result codes

noErr	0	No error

UTTrashBlocks

Use UTTrashBlocks to trash or free a range of cache blocks associated with a file.

```
pascal OSErr UTTrashBlocks(unsigned long beginPosition,
                           unsigned long byteCount,
                           VCBPtr volCtrlBlockPtr, short fileRefNum,
                           short tbOption)
```

beginPosition — Contains the beginning byte position of the blocks to trash. beginPosition must be a multiple of 512 bytes.

byteCount — Contains the number of bytes to trash. byteCount must be a multiple of 512 bytes.

volCtrlBlockPtr — Contains a VCBPtr that specifies the volume the file is on.

fileRefNum — Contains the file's reference number.

tbOption — Contains the trash block options. If zero, the range of blocks if trashed; If non-zero, the range of blocks is marked free.

UTTrashBlocks trashes or frees a range of cache blocks associated with the file specified by fileRefNum on the volume specified by volCtrlBlockPtr. The range is specified by beginPosition and byteCount. This routine is typically used when a file is shortened.

Result codes

noErr	0	No error

UTCacheReadIP

Use UTCacheReadIP to read a consecutive number of blocks into the caller's buffer.

```
pascal OSErr UTCacheReadIP(void *log2PhyProc,
                           unsigned long filePosition,
                           Ptr ioBuffer, short fileRefNum,
                           unsigned long reqCount,
                           unsigned long *actCount,
                           short cacheOption);
```

log2PhyProc — Contains a pointer to the logical to physical mapping routine.

filePosition — Contains the position in bytes to start reading from. This must fall on a 512-byte boundary (i.e., 0, 512, 1024, etc.).

ioBuffer — Contains a pointer to data buffer where the data will be read.

fileRefNum	Contains the file reference number.
reqCount	Contains the number of bytes to read. This must be a multiple of 512 bytes.
actCount	Contains a pointer to an unsigned long. UTCacheReadIP places the actual number of bytes read into the field referred to by this parameter .
cacheOption	Contains the cache options. The valid bits in the cacheOption parameter are: noCache please don't cache this read rdVerify always read blocks from disk (forced read)

UTCacheReadIP lets you request a multiple-block read from a file through the file system cache mechanism. UTCacheReadIP always reads a contiguous range of blocks. The number of contiguous blocks actually read is determined by the log2PhyProc and is returned in actCount.

The cacheOption parameter lets you request that reads not be cached, or that reads always come from the disk. A foreign file system handling a _Read call can simply use the ioPosMode word from the caller's parameter block for the cacheOption parameter.

Important note: UTCacheReadIP can potentially return an internal positive error result code. Because the file system should never return a positive result, your foreign file system should return ioErr if it cannot recover gracefully from one of the positive error results.

▲ **Warning:** UTCacheReadIP may only be called from a foreign file system handling a request through its HFSCIProc. Attempts to call UTCacheReadIP outside of the context of the HFSCIProc will cause a system crash. ▲

Result codes

noErr	0	No error
chNoBuf	1	Cache internal error - no free cache buffers (all reserved)
chInUse	2	Cache internal error - requested block is reserved
chNotfound	3	Cache internal error - requested block was not found (only if gbExist option)
ioErr	-36	Device I/O error
offLinErr	-65	Device was offline
notEnoughMemoryErr	-620	VM was not able to hold down enough physical memory

UTCacheWriteIP

Use UTCacheWriteIP to write a consecutive number of blocks from the caller's buffer.

```
pascal OSErr UTCacheWriteIP(void *log2PhyProc,
                            unsigned long filePosition, Ptr ioBuffer,
                            short fileRefNum, unsigned long reqCount,
                            unsigned long *actCount,
                            short cacheOption);
```

log2PhyProc	Contains a pointer to the logical to physical mapping routine.
filePosition	Contains the position in bytes to start writing to. This must fall on a 512-byte boundary (i.e., 0, 512, 1024, etc.).
ioBuffer	Contains a pointer to data buffer containing the data.
fileRefNum	Contains the file reference number.
reqCount	Contains the number of bytes to write. This must be a multiple of 512 bytes.
actCount	Contains a pointer to an unsigned long. UTCacheWriteIP places the actual number of bytes written into the field referred to by this parameter .
cacheOption	Contains the cache options. The valid bits in the cacheOption parameter are: noCache please don’t cache this write

UTCacheWriteIP lets you request a multiple-block write to a file through the file system cache mechanism. UTCacheWriteIP always writes a contiguous range of blocks. The number of contiguous blocks actually written is determined by the log2PhyProc and is returned in actCount.

The cacheOption parameter lets you request that writes not be cached. A foreign file system handling a _Write call can simply use the ioPosMode word from the caller’s parameter block for the cacheOption parameter.

Important note: UTCacheWriteIP can potentially return an internal positive error result code. Because the file system should never return a positive result, your foreign file system should return ioErr if it cannot recover gracefully from one of the positive error results.

▲ **Warning:** UTCacheWriteIP may only be called from a foreign file system handling a request through its HFSCIProc. Attempts to call UTCacheWriteIP outside of the context of the HFSCIProc will cause a system crash. ▲

Result codes

noErr	0	No error
chNoBuf	1	Cache internal error - no free cache buffers (all reserved)
chInUse	2	Cache internal error - requested block is reserved
chNotfound	3	Cache internal error - requested block was not found (only if gbExist option)
ioErr	-36	Device I/O error
offLinErr	-65	Device was offline
notEnoughMemoryErr	-620	VM was not able to hold down enough physical memory

UTBlockInFQHashP

Use UTBlockInFQHashP to see if a physical block on the volume is in a free cache block in the Macintosh volume cache.

```
pascal OSErr UTBlockInFQHashP(short vRefNum, unsigned long diskBlock);
```

vRefNum Contains the volume reference number.

diskBlock Contains the physical block number of the volume.

UTBlockInFQHashP lets you check to see if the specified block is in a free cache block in the Macintosh volume cache. UTBlockInFQHashP returns noErr if the disk block data is in a free cache block. A result of chNotfound indicates that the disk block is not in the free block queue (although it may in the cache as a reserved or released block).

Result codes

noErr	0	No error
chNotfound	3	Requested block was not found in the free block queue

4 THE DISK INITIALIZATION COMPONENT INTERFACE

ABOUT THIS CHAPTER

This chapter describes the Disk Initialization Package component interface, the data structures used by the Disk Initialization Package component interface, and the routines your foreign file system must provide to the Disk Initialization Package component interface.

To use this chapter, you should be familiar with the information in the chapter "The File System Manager" in this document, the information in *Inside Macintosh: Files*, the information in *Inside Macintosh: Devices*, and the information in the Technical Note "What Your Sony Drives for You".

ABOUT THE DISK INITIALIZATION PACKAGE COMPONENT INTERFACE

The Disk Initialization Package component interface allows a foreign file system to process Disk Initialization Package requests and allows a foreign file system to describe its disk initialization services to the File System Manager. Whenever a Disk Initialization Package request is made, the File System Manager will call the foreign file system through the Disk Initialization Package component interface to get additional information and request specific actions.

USING THE DISK INITIALIZATION PACKAGE COMPONENT INTERFACE

This section describes

- the Disk Initialization Package Component Interface record
- the Disk Initialization Package Component Interface request processing function and its function selectors

The Disk Initialization Package Component Interface Record

The Disk Initialization Package component interface record in an File System Descriptor record lets the foreign file system tell the Disk Initialization Package component interface how to dispatch Disk Initialization Package requests to the foreign file system. It tells the Disk Initialization Package component interface where the routine to handle Disk Initialization Package requests is in memory, the maximum length of volume names the foreign file system can accept, and the disk block size of your foreign file system's volume format. The data type DICIRec defines the Disk Initialization Package component interface record.

```
struct DICIRec {
     long              compInterfMask;     /* component flags */
     DICIUPP           compInterfProc;     /* pointer to request
                                              processing code */
     short             maxVolNameLength;   /* maximum length of your
                                              volume name */
     unsigned short    blockSize;          /* your file system's block
                                              size */
     long              reserved3;          /* --reserved, must be
                                              zero-- */
     long              reserved2;          /* --reserved, must be
                                              zero-- */
     long              reserved1;          /* --reserved, must be
                                              zero-- */
     };
     typedef struct DICIRec DICIRec;
     typedef DICIRec *DICIRecPtr;
```

Field descriptions

compInterfMask — Contains the Disk Initialization Package component interface dispatch mask. Currently the following bits are defined:

Bit		Meaning
0	diCILiveBit	If set, this file system is a candidate for the current formatting operation. This bit is maintained by the Disk Initialization Package component interface.
1-15		Reserved.
16	diCIDoesSparingBit	Set this bit if the foreign file system supports bad block disk sparing.
17	diCIHasMultiVolTypesBit	Set this bit if the foreign file system supports more than one volume type.
18	diCIHasExtFormatParamsBit	Set this bit if the foreign file system uses extended format parameters.
19-23		Reserved.
24-29		Reserved for the File System Manager's use.
30	fsmComponentBusyBit	If set, the FSM component interface is busy (i.e., one or more requests are outstanding). This bit is maintained by the component and used by the File System Manager to control the use of the SetFSInfo File System Manager service request.

	31	fsmComponentEnableBit	If set, the component may begin dispatching requests to the foreign file system. The foreign file system should set this bit with the SetFSInfo File System Manager service request once it is ready to receive requests.

compInterfProc Contains a pointer to the foreign file system's Disk Initialization Package component interface request processing function.

maxVolNameLength Contains the maximum length of volume names the foreign file system can accept.

blockSize Contains the size of the foreign file system disk block in bytes.

The Disk Initialization Package Component Interface Request Processing Function

The Disk Initialization Package Component Interface request processing function pointed to by compInterfProc in a DICIRec is called by the File System Manager to handle Disk Initialization Package requests. The Disk Initialization Package Component Interface request processing function must have the following form.

```
pascal OSErr DICIProc(short whatFunction, void *paramBlock,
                      void *fsdGlobalPtr);
```

whatFunction Contains a selector number which indicates what operation the foreign file system must perform.

paramBlock Contains a pointer to the parameter block passed to the Disk Initialization Package component interface request processing function. The contents of the parameter block are specific to the selector number passed.

fsdGlobalPtr Contains a pointer to the foreign file system's optional global data area.

When the foreign file system's DICIProc is called, it is passed a function selector number which specifies the operation to perform, a pointer to a buffer containing any additional parameters needed to process the operation, and a pointer to the foreign file system's global data area. The currently defined messages are listed in Table 4-1.

Table 4-1. Disk Initialization Package Component Interface Function Selectors

Message		Purpose
diCILoad	1	Load the disk initialization code and data resources (if they are not loaded or purged) and make them unpurgable.

diCIUnload	2	Make the disk initialization code and data resources purgable.
diCIEvaluateSizeChoices	3	Determine if the foreign file system can initialize a disk in the specified drive.
diCIExtendedZero	4	Initialize a disk (write an empty volume directory).
diCIValidateVolName	5	Determine if a volume name is valid.
diCIGetVolTypeInfo	6	If the foreign file system supports more than one volume type, provide a list of those volume types.
diCIGetFormatString	7	Supply strings used to build the Format pop-up menu and help balloons used in the disk initialization dialog box.
diCIGetExtFormatParams	8	Query the user (or another part of the system) for additional formatting information beyond that supplied by the disk initialization dialog box provided by the Disk Initialization Package.

How the foreign file system's DICIProc should handle each operation and what result codes should be returned are described in the following sections.

Note: The DICIProc cannot be called at interrupt time. Thus, it may make Memory Manager requests and synchronous operating system requests.

diCILoad

The diCILoad function selector tells the foreign file system to load the disk initialization code and data resources (if they are not loaded or purged) and make them unpurgable. The paramBlock parameter is not used with the diCILoad function selector.

Result codes

noErr	0	No error

any of the result codes returned by the Resource Manager, GetFSInfo, or SetFSInfo

diCIUnload

The diCIUnload function selector tells the foreign file system to make the disk initialization code and data resources purgable. The paramBlock parameter is not used with the diCIUnload function selector.

Result codes

noErr	0	No error

diCIEvaluateSizeChoices

The diCIEvaluateSizeChoices function selector allows a foreign file system to determine if it can initialize a disk in the specified drive. If the disk drive can format disks in more than one size, the foreign file system indicates what sizes it can support and the preferred size. The paramBlock parameter points to a DICIEvaluateSizeRec record.

DICIEvaluateSizeRec

←	defaultSizeIndex	short	The foreign file system returns the index number of the preferred or default SizeListRec in this field; the first SizeListRec is at index position 1. If none of the SizeListRec records contain a valid size choice for the foreign file system, return zero.
→	numSizeEntries	short	Contains the number of SizeListRec records in the array pointed to by sizeListPtr.
→	driveNumber	short	Contains the drive number.
→	sizeListPtr	SizeListRecPtr	Contains a pointer to an array of SizeListRec records.
→	sectorSize	unsigned short	Contains the number of bytes per sector (sectors may not be the same size as blocks used by a foreign file system). Use this value to help determine if the foreign file system can use disks mounted on this drive.

The sizeListPtr field in the DICIEvaluateSizeRec points to an array of SizeListRec records.

SizeListRec

←	sizeListFlags	short	The foreign file system returns the size list flags in this field. All bits are currently reserved except: diCISizeListOKBit — Set this bit if the size specified in sizeEntry can be used by this foreign file system.
→	sizeEntry	FormatListRec	Contains a FormatListRec which specifies a possible size of the drive.

The sizeEntry field in a SizeListRec is a FormatListRec.

FormatListRec

→	volSize	unsigned long	Contains the disk capacity in disk sectors (sectors may not be the same size as blocks used by a foreign file system).

→	formatFlags	SignedByte	Contains the format flags, density bit, and number of disk sides. The sides, sectorsPerTrack, and tracks are valid if diCIFmtFlagsValidBit is set. The current disk has this format if diCIFmtFlagsCurrentBit is set. The disk is double-density (.SONY driver only) if diCIFmtFlagsDoubleDensityBit is set. The low nibble of the byte contains the number of disk sides.
→	sectorsPerTrack	SignedByte	Contains the average number of sectors per track.
→	tracks	unsigned short	Contains the number of disk tracks.

The array of SizeListRec records is built from information obtained from the disk driver. Specifically, the File System Manager obtains the SizeListRec records by calling the disk driver's _Status function with csCode set to Return Format List (6). If the disk driver does not support the Return Format List request, a single SizeListRec is constructed from the information in the drive queue element.

The foreign file system must index through the array of SizeListRec records. The number of elements in the array is indicated by numSizeEntries. If the foreign file system can format the disk using the size specified by a particular SizeListRec, it sets the diCISizeListOKBit in the SizeListRec's sizeListFlags field. SizeListRec records usable by the foreign file system are shown in the Format pop-up menu in the disk initialization dialog box.

The index of the preferred SizeListRec is returned in the DICIEvaluateSizeRec record's defaultSizeIndex field where defaultSizeIndex is in the range 1 to numSizeEntries. This is the menu item shown in the Format pop-up menu in the disk initialization dialog box if the current disk is already owned by this foreign file system.

Result codes

noErr	0	No error

diCIExtendedZero

The diCIExtendedZero function selector allows a foreign file system to initialize a disk. The paramBlock parameter points to a DICIExtendedZeroRec record.

DICIExtendedZeroRec

→	driveNumber	short	Contains the drive number of the device which should be initialized with an empty volume directory structure.
→	volNamePtr	StringPtr	Contains a pointer to the name that the volume should be given.
→	fsid	short	Contains the target file system ID.

→	volTypeSelector	short	Contains the volume type selector if the foreign file system supports more than one volume type (diCIHasMultiVolTypesBit is set in the compInterfMask field of the DICIRec).
→	numDefectBlocks	unsigned short	Contains the number of bad disk blocks or zero. The block size is the size specified in the blockSize field of the DICIRec.
→	defectListSize	unsigned short	Contains the size of the defect list buffer if numDefectBlocks is not zero. For example, if the volume size is 800K and your file system's block size is 1024 bytes, then defectListSize = 100 (800 bits = 100 bytes) and the defect list buffer is 100 bytes in size.
→	defectListPtr	Ptr	Contains a pointer to the defect list buffer if numDefectBlocks is not zero. The buffer contains a bitmap of the disk blocks. A zero bit means the block is good, and a one bit means the block is defective. This buffer is allocated by the Disk initialization Package and is released by the Disk initialization Package after returning from this request.
→	volSize	unsigned short	Contains the size of the volume in disk sectors (sectors may not be the same size as blocks used by a foreign file system).
→	sectorSize	unsigned short	Contains the number of bytes per disk sector (sectors may not be the same size as blocks used by a foreign file system).
→	extendedInfoPtr	Ptr	Contains a pointer to the foreign file system's extended formatting information (if diCIHasExtFormatParamsBit is set in the compInterfMask field of the DICIRec), or nil.

The foreign file system must initialize the disk specified by driveNumber.

If the foreign file system has set diCIHasMultiVolTypesBit in the compInterfMask field of the DICIRec indicating that it supports multiple volume formats, the volume type selected is in volTypeSelector.

If the foreign file system has set diCIDoesSparingBit in the compInterfMask field of the DICIRec, it can use the information in numDefectBlocks, defectListSize, and defectListPtr to perform bad block sparing.

Result codes

noErr	0	No error
diCICriticalSectorBadErr	20	Disk cannot be spared; sectors critical to the volume format are bad
diCISparingFailedErr	21	Disk cannot be spared; general error
diCITooManyBadSectorsErr	22	Disk cannot be spared; too many bad sectors were found
diCIUnknownVolTypeErr	23	The volume type passed is not supported
diCINoExtendInfoErr	27	extendedInfoPtr was required but nil was passed
paramErr	-50	The file system ID passed does not belong to this foreign file system

diCIValidateVolName

The diCIValidateVolName function selector allows a foreign file system to determine if a volume name is valid. The paramBlock parameter points to a DICIValidateVolNameRec record.

DICIValidateVolNameRec

→	theChar	char	Contains the character to validate.
←	hasMessageBuffer	Boolean	The foreign file system returns true in this field if it is providing its own alert message when an illegal character is detected (messageBufferPtr must not be nil).
→	charOffset	short	Contains the position of the current character (the first character is at position 1).
↔	messageBufferPtr	StringPtr	If messageBufferPtr is not nil, then it points to a Str255 buffer where the foreign file system can return its own alert message when an illegal character is detected.
→	charByteType	short	Contains the identity of theChar as a 1-byte character or as the first or second byte of a 2-byte character. The values can be: smFirstByte First byte of a 2-byte character smSingleByte 1-byte character smLastByte Second byte of 2-byte character smFirstByte, smSingleByte, and smLastByte are defined in Script.h.

Foreign file systems must be able to determine if a volume name is valid. The volume name must be validated:

- as characters are entered into the name field in the disk initialization dialog box. This gives the foreign file system an opportunity to examine the syntax of the name as it is being assembled.

- when the volume format selected in the Format pop-up menu in the disk initialization dialog box changes. This gives the foreign file system an opportunity to examine the syntax of the name already entered in the the name field.
- when characters are pasted into the name field of the disk initialization dialog box. This gives the foreign file system an opportunity to examine the syntax of the characters added to the name.

If an illegal character is detected, the foreign file system must return paramErr.

If messageBufferPtr is not nil, then it points to Str255 buffer where the foreign file system can return its own alert message when an illegal character is detected. If an illegal character alert message is supplied, the foreign file system must return true in the hasMessageBuffer field. If hasMessageBuffer is left false, the Disk Initialization Package uses the default illegal character alert message. The messageBufferPtr field is set to nil (no buffer is provided) when the volume format selected in the Format pop-up menu in the disk initialization dialog box changes and when characters are pasted into the name field of the disk initialization dialog box.

The Disk Initialization Package ensures volume names are not too long.

Result codes

noErr	0	No error
paramErr	-50	The character passed is illegal in this foreign file system's volume names

diCIGetVolTypeInfo

The diCIGetVolTypeInfo function selector allows a foreign file system that supports more than one volume type to provide a list of those volume types. The paramBlock parameter points to a DICIGetVolTypeInfoRec record.

DICIGetVolTypeInfoRec

→	volSize	unsigned long	Contains the disk capacity in disk sectors (sectors may not be the same size as blocks used by a foreign file system).
→	sectorSize	unsigned short	Contains the number of bytes per sector (sectors may not be the same size as blocks used by a foreign file system).
←	numVolTypes	short	The foreign file system returns the number of volume types it supports in this field.
←	volTypesBuffer[4]	Str32	The foreign file system returns the names of volume types it supports in these string fields.

Some file systems support more than one volume type, for example Macintosh File System supports both HFS and MFS volume types. If your foreign file system supports more than one volume type, it should set the diCIHasMultiVolTypesBit in the compInterfMask field of its DICIRec. If the diCIHasMultiVolTypesBit is not set, the Disk Initialization Package Component Interface request processing function will not be called with the diCIGetVolTypeInfo function selector.

The volSize and sectorSize fields are provided to help the foreign file system determine which of its volume types can be used.

The foreign file system returns the number of volume types it supports in the numVolTypes field. Up to four volume types are allowed.

The name of each supported volume type is returned in one of the four volTypeBuffer string buffers. These strings are used, together with the file system name and disk size, to compose the menu item strings to be displayed in the disk initialization dialog's Format pop-up menu. The order of the strings is important because the ordering number will be passed as the volTypeSelector to the foreign file system when it is called with the diCIExtendedZero function selector. The value of volTypeSelector is only meaningful to your file system as it is a direct mapping to the types returned by this function request in the volTypesBuffer string buffers.

Your foreign file system's default volume type string must be returned in volTypesBuffer[1] for that volume type to be selected as the default in the Format pop-up menu in the disk initialization dialog box.

Your volume type information and the corresponding type selectors should be published in your foreign file system's documentation so that specific volume types can be initialized programmatically by applications with the DIXZero function.

Result codes

noErr	0	No error

diCIGetFormatString

The diCIGetFormatString function selector allows a foreign file system to supply strings used to build the Format pop-up menu and help balloons used in the disk initialization dialog box. The paramBlock parameter points to a DICIGetFormatStringRec record.

DICIGetFormatStringRec

→ volSize	unsigned long	Contains the disk capacity in disk sectors (sectors may not be the same size as blocks used by a foreign file system).
→ sectorSize	unsigned short	Contains the number of bytes per sector (sectors may not be the same size as blocks used by a foreign file system).

→	volTypeSelector	short	Contains the volume type selector if the foreign file system supports more than one volume type (diCIHasMultiVolTypesBit is set in the compInterfMask field of the DICIRec).
→	stringKind	short	Contains the kind of string requested. The kinds are: diCIAlternateFormatStr get the alternate format string diCISizePresentationStr get the size presentation string
←	stringBuffer	Str255	The foreign file system returns the string requested in this field.

A foreign file system can supply strings used to build the Format pop-up menu and help balloons used in the disk initialization dialog box. If the foreign file system doesn't provide the string, it can return diCIUnknownDICallErr and Disk Initialization Package will use its default strings.

If stringKind is diCIAlternateFormatStr, the foreign file system can provide a string that will be used augment the help balloon displayed for a volume type in the Format pop-up menu in the disk initialization dialog box. For example, without the alternate format string, the balloon text for an Macintosh 800K disk reads "To erase the selected disk and change it to a Macintosh 800K disk, choose this item, then click Erase." With the alternate format string " (also known as a double-sided Macintosh disk)", the balloon text for an Macintosh 800K disk reads "To erase the selected disk and change it to a Macintosh 800K disk (also known as a double-sided Macintosh disk), choose this item, then click Erase."

If stringKind is diCISizePresentationStr, the foreign file system can provide a string that replaces the volume type in the Format pop-up menu in the disk initialization dialog box. For example, by supplying the size presentation string, the volume type string in the Format pop-up menu could be changed from "Macintosh 720K" to "Macintosh HFS Interchange Format".

Result codes

noErr	0	No error
diCIUnknownDICallErr	25	The requested string isn't provided by this foreign file system

diCIGetExtFormatParams

The diCIGetExtFormatParams function selector allows a foreign file system a chance to query the user (or another part of the system) for additional formatting information beyond that supplied by the disk initialization dialog box provided by the Disk Initialization Package. The paramBlock parameter points to a DICIGetExtendedFormatRec record.

DICIGetExtendedFormatRec

→	driveNumber	short	Contains the drive number of the device to be initialized.

→	volTypeSelector	short	Contains the volume type selector if the foreign file system supports more than one volume type (diCIHasMultiVolTypesBit is set in the compInterfMask field of the DICIRec).
→	volSize	unsigned long	Contains the disk capacity in disk sectors (sectors may not be the same size as blocks used by a foreign file system).
→	sectorSize	unsigned short	Contains the number of bytes per sector (sectors may not be the same size as blocks used by a foreign file system).
→	fileSystemSpecPtr	FSSpecPtr	Contains a pointer to foreign file system's FSSpec.
←	extendedInfoPtr	Ptr	The foreign file system returns a pointer to the extended formatting information in this field.

A foreign file system may need more information from the user (or another part of the system) than can be provided by the disk initialization dialog box. The diCIGetExtFormatParams function selector allows a foreign file system a chance to query the user for extended formatting information after the user selects the Erase or Initialize button in the disk initialization dialog box. The diCIGetExtFormatParams function selector is called if the diCIHasExtFormatParamsBit is set in the compInterfMask field of the foreign file system's DICIRec.

When called with the diCIGetExtFormatParams function selector, the foreign file system should allocate memory (if it hasn't done so when called with the diLoad function selector) for the extended formatting information structure, open its resource fork using the FSSpec passed in the fileSystemSpecPtr field, display one or more dialogs to collect the extended information, close its resource fork, and then return a pointer to the extended formatting information in the extendedInfoPtr field. When called with the diUnload function selector, the foreign file system should dispose of its extended formatting information structure.

Any dialogs displayed should contain both a "Continue" and a "Cancel" button with the "Continue" button as the default selection. If the "Continue" button is selected, noErr should be returned as the result along with the extended formatting information. If the "Cancel" button is selected, diCIUserCancelErr should be returned as the result.

The extended formatting information data structure used by your foreign file system should be documented so that application programs can pass the information programmatically in the DIXZero request.

Result codes

noErr	0	No error
diCIUserCancelErr	1	The "Cancel" button was selected by the user

APPENDIX A: THE EXTENDED DISK INITIALIZATION PACKAGE

ABOUT THIS APPENDIX

This appendix documents three Disk Initialization Package functions not found in *Inside Macintosh: Files*

APPLICATION PROGRAM INTERFACE

The existing application program interface to the Disk Initialization Package as described in *Inside Macintosh: Files*will continue to be supported by the enhanced Disk Initialization Package. Applications which wish to initialize *only* Macintosh disks will continue to work and will require no changes. However, if an application wants to initialize non-Macintosh disks, it must use the new extended DIXFormat and DIXZero calls as described below.

Extended programmatic interfaces to media formatting and volume initialization functions are required such that applications may specify additional information for the overall formatting operation. This information corresponds to the new items in the user interface described in the previous section: file system type and disk size. The extended programmatic interface adds three new functions to the Disk Initialization Package called DIXFormat and DIXZero (for extended DIFormat and DIZero), and DIReformat.

> **Warning:** Applications should insure that the extended Disk Initialization Package functions are present before making the DIXFormat, DIXZero, or DIReformat calls. This is done by calling Gestalt with the gestaltFSAttr selector. The extended Disk Initialization Package functions is available if the Gestalt function returns a result of noErr and the gestaltHasExtendedDiskInitbit (bit 6) is set in the response parameter. Due to the nature of older versions of the Disk Initialization Package, making the extended requests when they are not available may cause a system crash.

Listing 5-1 illustrates how you use Gestalt to determine if the extended Disk Initialization Package functions are available.

Listing 5-1. Testing for extended Disk Initialization Package functions

```
Boolean  HasExtendedDIFunctions(void)
{
   long response;

   if (Gestalt(gestaltFSAttr, &response) == noErr)
      return ((response & (1L << gestaltHasExtendedDiskInit)) != 0);
   else
      return (false);
}
```

DIXFormat

The DIXFormat function performs the same function as the DIFormat function except that drive size may be specified.

```
pascal OSErr DIXFormat(short drvNum, Boolean fmtFlag,
                       unsigned long fmtArg, unsigned long *actSize);
```

drvNum	Contains the driver number of the drive to format.
fmtFlag	Contains a boolean value which specifies the meaning of the fmtArg paramter.
fmtArg	If fmtFlag is true, fmtArg specifies the actual value to be passed to the disk driver in the csParam field of the parameter block when the "format" _Control call is made to initialize the disk media. (The value is an index into the size list. For an explanation of appropriate values for this parameter, see the Technical Note "What Your Sony Drives For You".) If fmtFlag is false, fmtArg specifies the desired size of the media in number of 512-byte blocks. The disk driver is called to get possible sizes and the values in an to attempt to match the requested size. If more than one size list entry exists for the *same size*, the first entry in the list returned by the driver that best matches the fmtArg parameter will be used. For more information about the size list, see the Technical Note "What Your Sony Drives For You". If the specified size is larger than the largest size in the size list returned by the driver, then the largest size will be used and that size is returned in actSize. If the specified size is smaller than the smallest size in the size list returned by the driver, then the smallest size will be used and that size is returned in actSize. For a specified value that is in between and without an exact match, the value closest to and smaller than the requested size is used.
actSize	Contains a pointer to an unsigned long. Upon completion of a successful formatting operation, DIXFormat places the actual size of the formatted media in number of 512-byte blocks into the field referred to by this parameter.

The formatting of file systems requiring specific media formats should be done by specifying those media formats explicitly and not by counting on disk size alone. Foreign file systems with specific media requirements should use the driver specific information in the size list or should make appropriate driver _Status calls for additional information when called upon to "evaluate the size list".

As in DIFormat, DIXFormat does not unmount the volume. You have to unmount the volume before issuing this call if necessary. If the volume has not been unmounted, then DIXFormat will return volOnLinErr error.

Result codes

noErr	0	No error
volOnLinErr	-55	Volume is online
lastDskErr	-64	Last of the range of low-level disk errors
…		
firstDskErr	-84	First of the range of low-level disk errors

DIXZero

The DIXZero function performs the same function as the DIZero function except that the file system, format result, volume type, volume size and extended formatting information may be specified.

```
pascal OSErr DIXZero(short drvNum, ConstStr255Param volName, short fsid,
                     short mediaStatus, short volTypeSelector,
                     unsigned long volSize, void *extendedInfoPtr);
```

drvNum — Contains the driver number of the drive to initialize.

volName — Contains a pointer to a Pascal string which specifies the name of the volume.

fsid — Contains the ID of the file system whose format should be written to the disk. The file system ID can be obtained using the File System Manager GetFSInfo function.

mediaStatus — Contains a flag to indicate the status of the disk media. Its value is the result code returned from the DIVerify function. If mediaStatus is non-zero, then the disk contains bad sectors and needs to be spared. If the file system specified doesn't support bad block sparing (the diCIDoesSparingBit in the compInterfMask field of the DICIRec is clear), the Disk Initialization Package will just return this value as the function result. If the file system supports bad block sparing, then the Disk Initialization Package will gather the defect list and pass it to the file system.

volTypeSelector — Contains the volume type selector if the foreign file system supports more than one volume type (diCIHasMultiVolTypesBit is set in the compInterfMask field of the file system's DICIRec).

volSize — Contains the size in 512-byte blocks of the file system that should be written to the drive specified by drvNum. This is the size returned in the actSize field by DIXFormat - the amount of space usable by a file system on the specified drive as it is currently formatted. If the specified size doesn't match with the current disk format size, DIXZero will return diCIVolSizeMismatchErr.

fsParams — Contains a pointer to the foreign file system's extended formatting information (if diCIHasExtFormatParamsBit is set in the compInterfMask field of the DICIRec), or nil.

As in DIZero, DIXZero does not unmount the volume but it will, however, mount the volume if the operation is successful. You have to unmount the volume before issuing this call if necessary. If the volume is mounted when DIZero or DIXZero is called, then a volOnLinErr error will be returned.

Result codes

noErr	0	No error
ioErr	-36	I/O error
paramErr	-50	Drive number specified is bad
volOnLinErr	-55	Volume is already online
nsDrvErr	-56	No such drive
lastDskErr	-64	Last of the range of low-level disk errors
…		
firstDskErr	-84	First of the range of low-level disk errors
memFullErr	-108	Not enough memory

DIReformat

The DIReformat function reformats disk volume.

```
pascal OSErr DIReformat(short drvNum, short fsid,
                        ConstStr255Param volName,
                        ConstStr255Param msgText);
```

drvNum — Contains the driver number of the drive to format.

fsid — Contains the ID of the file system whose format should be written to the disk. The file system ID can be obtained using the File System Manager GetFSInfo function.

volName — Contains a pointer to a Pascal string which specifies the name of the volume.

msgText — Contains a pointer to a Pascal string which specifies the explanatory text to be displayed in the disk initialization dialog box.

In the past, reformatting disk was accomplished by calling the DIBadMount function with the high word of the evtMessage parameter set to noErr and the explanatory text was set with the ParamText function. The DIReformat function provides the caller the ability to provide the explanatory text, the default file system ID, and the default name for the reformatted disk.

Note: The volume in the drive specified by drvNum *must be* mounted when calling DIReformat .

Result codes

noErr	0	No error
diCINoMessageTextErr	28	msgText was not provided
paramErr	-50	Drive number specified is bad
nsDrvErr	-56	No such drive
lastDskErr	-64	Last of the range of low-level disk errors
. . .		
firstDskErr	-84	First of the range of low-level disk errors
memFullErr	-108	Not enough memory

APPENDIX B: ADDITIONAL FILE MANAGER ROUTINES

ABOUT THIS APPENDIX

This appendix documents two File Manager routines that are not in *Inside Macintosh: Files* but must be documented for foreign file systems.

APPLICATION PROGRAM INTERFACE

The following two functions, PBHUnmountVol and PBGetXCatInfo are not documented in *Inside Macintosh: File*s Foreign file systems must respond correctly to PBHUnmountVol and may want to support PBGetXCatInfo. Applications may use PBGetXCatInfo, but must *never* call PBHUnmountVol.

▲ **Warning:** PBHUnmountVol is reserved for use only by the Macintosh operating system. The operating system uses PBHUnmountVol to unconditionally unmount volumes before system shutdown or restart. Applications should never call PBHUnmountVol because PBHUnmountVol can close files in use by other applications which can cause data loss or media corruption. ▲

PBHUnmountVol

The PBHUnmountVol function unconditionally unmounts a volume

```
pascal OSErr PBHUnmountVol(ParmBlkPtr paramBlock);
```

paramBlock Contains a pointer to a ParamBlockRec.

ParamBlockRec

←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	short	A volume reference number, a working directory reference number, drive number, or 0 for the default volume.

The PBHUnmountVol function unconditionally unmounts the specified volume. All user files on the volume will be closed by the file system owning the volume.

The PBHUnmountVol function always executes synchronously.

▲ **Warning:** PBHUnmountVol is reserved for use only by the Macintosh operating system. The operating system uses PBHUnmountVol to unconditionally unmount volumes before system shutdown or restart. Applications should never call PBHUnmountVol because PBHUnmountVol can close files in use by other applications which can cause data loss or media corruption. ▲

ASSEMBLY-LANGUAGE INFORMATION

The trap word for PBHUnmountVol is A20E.

Result codes

noErr	0	No error
nsvErr	–35	No such volume reference number
ioErr	–36	I/O error
bdNamErr	–37	Bad volume name
paramErr	–50	No default volume
nsDrvErr	–56	No such drive
extFSErr	–58	No file system claimed the volume

PBGetXCatInfo

You can use the PBGetXCatInfo function to get the short name (MS-DOS format name) and ProDOS information for files and directories.

```
pascal OSErr PBGetXCatInfoSync(XCInfoPBPtr paramBlock);
pascal OSErr PBGetXCatInfoAsync(XCInfoPBPtr paramBlock);
```

paramBlock Contains a pointer to a XCInfoPBRec.

XCInfoPBRec

→	ioCompletion	StringPtr	Contains a pointer to PBGetXCatInfoAsync's completion routine.
←	ioResult	OSErr	PBGetXCatInfo places its result code into this field.
→	ioNamePtr	StringPtr	Contains a pointer to the object name, or nil when ioDirID specifies a directory that's the object.
→	ioVRefNum	short	Contains a volume specification.
→	ioShortNamePtr	OSErr	Contains a pointer to a Pascal string buffer (minimum 13 bytes). PBGetXCatInfo places the short name into the field referred to by this parameter. ioShortNamePtr cannot be nil.
←	ioPDType	short	PBGetXCatInfo places the ProDOS file type into this field.
←	ioPDAuxType	long	PBGetXCatInfo places the ProDOS auxiliary type into this field.
→	ioDirID	long	Contains a directory ID.

PBGetXCatInfo returns the short name (MS-DOS format name) and ProDOS file/auxiliary type information for files and directories on volumes that support this function. Volumes that support PBGetXCatInfo will have the bHasShortName bit set in the vMAttrib field returned by PBHGetVolParms.

The following data structure and inline glue code is needed for applications that wish to use PBGetXCatInfo.

```
  struct XCInfoPBRec {
       QElemPtr         qLink;
       short            qType;
       short            ioTrap;
       Ptr              ioCmdAddr;
       ProcPtr          ioCompletion;     /* A pointer to a completion
                                             routine */
       OSErr            ioResult;         /* The result code of the
                                             function */
       StringPtr        ioNamePtr;        /* Pointer to object name or
                                             nil */
       short            ioVRefNum;        /* A volume specification */
       long             filler1;
       StringPtr        ioShortNamePtr;   /* A pointer to the short name
                                             string buffer - required! */
       short            filler2;
       short            ioPDType;         /* The ProDOS file type */
       long             ioPDAuxType;      /* The ProDOS aux type */
       long             filler[2];
       long             ioDirID;          /* A directory ID */
       };
       typedef struct XCInfoPBRec XCInfoPBRec;
       typedef XCInfoPBRec *XCInfoPBPtr;

#pragma parameter __D0 PBGetXCatInfoSync(__A0)
pascal OSErr PBGetXCatInfoSync(XCInfoPBPtr paramBlock)
               = {0x703A,0xA260};
#pragma parameter __D0 PBGetXCatInfoAsync(__A0)
pascal OSErr PBGetXCatInfoAsync(XCInfoPBPtr paramBlock)
               = {0x703A,0xA660};
```

For more information about short names and ProDOS file/auxiliary types, see *Inside AppleTalk, second edition* , Chapter 13 AppleTalk Filing Protocol, and the *Apple II File Type Notes*.

ASSEMBLY-LANGUAGE INFORMATION

The PBGetXCatInfo function uses routine selector $003A of HFSDispatch.

Result codes

noErr	0	No error
nsvErr	-35	No such volume
fnfErr	-43	File not found
paramErr	-50	Function not supported by volume
dirNFErr	-120	Directory not found

APPENDIX C: FILE SYSTEM ROUTINE TABLE

ABOUT THIS APPENDIX

This appendix lists all File Manager routine and the select codes that your foreign file system's HFSCIProc can be called with.

Routine Name	Select Code	Required	Volume Attributes	Notes
File Manager Traps				
Open	$A000	•		
Close	$A001	•		
Read	$A002	•		
Write	$A003	•		
GetVolInfo	$A007	•		
Create	$A008	•		
Delete	$A009	•		
OpenRF	$A00A	•		
Rename	$A00B	•		
GetFileInfo	$A00C	•		
SetFileInfo	$A00D	•		
UnmountVol	$A00E	•		
MountVol	$A00F	•		
Allocate	$A010	•		
GetEOF	$A011	•		
SetEOF	$A012	•		
FlushVol	$A013	•		
GetVol	$A014	•		
SetVol	$A015	•		
FInitQueue	$A016			Never passed to foreign file systems.
Eject	$A017	•		
GetFPos	$A018	•		
Offline	$A035	•		
SetFilLock	$A041	•		
RstFilLock	$A042	•		
SetFilType	$A043	•		This call is obsolete, but it does get passed to foreign file systems. You should return paramErr.
SetFPos	$A044	•		
FlushFile	$A045	•		
HFS Calls				
HOpen	$A200	•		
HGetVInfo	$A207	•		
HCreate	$A208	•		
HDelete	$A209	•		
HOpenRF	$A20A	•		
HRename	$A20B	•		

HGetFileInfo	$A20C	•		
HSetFileInfo	$A20D	•		
HUnmountVol	$A20E	•		Warning: reserved for system use only at system shutdown or restart! HUnmount unmounts a volume, even if files are open on the volume. The foreign file system should flush and close all open files before unmounting the volume.
AllocContig	$A210	•		
HGetVol	$A214	•		
HSetVol	$A215	•		
HSetFLock	$A241	•		
HRstFLock	$A242	•		
HFS Dispatch Calls				
OpenWD	$0001	•		
CloseWD	$0002	•		
CatMove	$0005	•		
DirCreate	$0006	•		
GetWDInfo	$0007	•		
GetFCBInfo	$0008	•		
GetCatInfo	$0009	•		
SetCatInfo	$000A	•		
SetVolInfo	$000B	•		
LockRng	$0010	•		
UnlockRng	$0011	•		
CreateFileIDRef	$0014		bHasFileIDs	
DeleteFileIDRef	$0015		bHasFileIDs	
ResolveFileIDRef	$0016		bHasFileIDs	
ExchangeFiles	$0017		bHasFileIDs	
CatSearch	$0018		bHasCatSearch	
OpenDF	$001A			Never passed to foreign file systems. The foreign file system will only see Open and HOpen.

MakeFSSpec	$001B			If your foreign file system doesn't handle MakeFSSpec, the Macintosh file system will make the FSSpec by determining the real vRefNum and real parent dirID, and then indexing through the parent directory until a matching name is found. This can be *very* slow if the directory has a lot of entries or if your foreign file system implementation of GetCatInfo is slow. If you don’t support MakeFSSpec, return paramErr.
Desktop Manager Calls				
DTGetPath	$0020		bHasDesktopMgr	The returned ioDTRefNum must be a file reference number (a FCB must be used).
DTCloseDown	$0021		bHasDesktopMgr	
DTAddIcon	$0022		bHasDesktopMgr	
DTGetIcon	$0023		bHasDesktopMgr	
DTGetIconInfo	$0024		bHasDesktopMgr	
DTAddAPPL	$0025		bHasDesktopMgr	
DTRemoveAPPL	$0026		bHasDesktopMgr	
DTGetAPPL	$0027		bHasDesktopMgr	
DTSetComment	$0028		bHasDesktopMgr	
DTRemoveComment	$0029		bHasDesktopMgr	
DTGetComment	$002A		bHasDesktopMgr	
DTFlush	$002B		bHasDesktopMgr	Do nothing (noErr) on remote shared volumes.
DTReset	$002C		bHasDesktopMgr	Do nothing (noErr) on remote shared volumes.
DTGetInfo	$002D		bHasDesktopMgr	Return paramErr on remote shared volumes.
DTOpenInform	$002E		bHasDesktopMgr	The returned ioDTRefNum must be a file reference number (a FCB must be used). Return paramErr on remote shared volumes.
DTDelete	$002F		bHasDesktopMgr	Return paramErr on remote shared volumes.

AppleShare Calls				
GetVolParms	$0030	• (see Note)		GetVolParms is required if you want to support any of the calls that require a volume attribute bit.
GetLogInInfo	$0031		bAccessCntl	
GetDirAccess	$0032		bAccessCntl	
SetDirAccess	$0033		bAccessCntl	
MapID	$0034		bAccessCntl	
MapName	$0035		bAccessCntl	
CopyFile	$0036		bHasCopyFile	
MoveRename	$0037		bHasMoveRename	
OpenDeny	$0038		bHasOpenDeny	
OpenRFDeny	$0039		bHasOpenDeny	
GetXCatInfo	$003A		bHasShortName	
GetVolMountInfoSize	$003F			
GetVolMountInfo	$0040			
VolumeMount	$0041			
Share	$0042			Handled by AppleShare or File Sharing server.
UnShare	$0043			Handled by AppleShare or File Sharing server.
GetUGEntry	$0044			Handled by AppleShare or File Sharing server.
GetForeignPrivs	$0060			
SetForeignPrivs	$0061			

APPENDIX D: FILE SYSTEM MANAGER QUESTIONS AND ANSWERS

ABOUT THIS APPENDIX

This appendix contains a bunch of questions and answers gathered by Developer Support that will be useful to developers of foreign file systems.

QUESTIONS AND ANSWERS

Q: If my foreign file system doesn't support one of the non-required HFSDispatch select codes, what should it return?

A: The foreign file system should return paramErr (-50) if it doesn't support one of the non-required HFSDispatch select codes.

Q: I'm implementing the Desktop Manager calls. How big can icons get?

A: Apple's Desktop Manager allows icons to be up to 4500 bytes, you should too.

Q: What fields in an FCBRec can be used by a foreign file system for its own purposes?

A: See Chapter 3, The File System Utility Routines, of the Guide to the File System Manager. It tells what fields can be used by your foreign file system and tells how those fields are used by the HFS file system.

Q: What are the catalog node ID (CNID) numbers below fsUsrCNID (16) used for?

A: CNID 1 is reserved for the parent ID of the root directory (fsRtParID) and CNID 2 is reserved for the directory ID of the root directory (fsRtDirID). On HFS volumes, 3 is used for the extents file, 4 is used for the catalog file, and 5 is used for the bad allocation block file. Your foreign file system can use CNIDs 5 through 15 for whatever purposes it needs.

When _UnmountVol is called, the File Manager checks to see if there are any open files on that volume with a file number (fcbFlNm in the FCBRec) of fsUsrCNID or greater. If so, the File Manager does not pass the _UnmountVol request on to the file system that owns the volume and a result of fBsyErr is returned to the caller. You can use CNIDs 5 through 15 for files opened by your foreign file system that should not prevent _UnmountVol from succeeding (for example, you should use a CNID below fsUsrCNID for the file number in FCBRecs used for the Desktop Manager DTGetPath or DTOpenInform routines).

Q: What are the advantages of supporting the Desktop Manager select codes?

A: If you don’t support the Desktop Manager select codes, the Finder will attempt to create a Desktop file to keep track of application mappings and icons. The advantages of the Desktop Manager over the Desktop file are:

- There is a documented interface for the Desktop Manager functions that developers can use.
- The Desktop Manager can be used on shared volumes. The Desktop file, since it is simply a resource file, can only allow write access to one user or process - it cannot be shared. Under System 7, that process is the Finder.
- The Desktop Manager functions can be implemented by an external file system in the most efficient way possible for that particular file system.
- The Desktop Manager doesn't have the file size limitations imposed by the Resource Manager; 2727 resources and approximately 16Mb of resource data.

Q: How does information get removed from the Desktop file or Desktop Database?

A: With both the Desktop file and the Desktop Manager database, icon data is not removed from the database unless the Desktop database or file is rebuilt from scratch. That's because the Finder (and file system for that matter) has no way of being notified when the last file of a given creator and type is removed from a volume. If the Finder removes an application (a file with type APPL), then the Finder attempts to remove the application mapping information for that copy of the application from the Desktop database or file.

Q: We want to provide the shared desktop database environment. What minimal set of calls do we need to support for this?

A: It really depends on how your file system is going to be used. Apple's read-only foreign file systems such as ISO 9660 let you open the desktop database, get info on it (all zeros are returned), and get comments on the root directory. All other Desktop Manager routines return an error. Apple's two read/write file systems that support the Desktop Manager, AppleShare and HFS, both support all of the original Desktop Manager calls (selectors $20 through $2A). HFS supports the selectors $2B through $2F; DTFlush, DTReset, DTGetInfo, DTOpenInform, and DTDelete don't make sense in a shared environment. So, I'd say you need to support selectors $20 through $2A and then return an appropriate error for the others. AppleShare returns noErr and does nothing for DTFlush and DTReset and returns paramErr for DTGetInfo, DTOpenInform, and DTDelete.

Q: We do not necessarily want full AppleShare (AFP) access-control, but we need more information on foreign file privilege models.

A: To be compatible with existing Macintosh applications, you must mimic the HFS file system's permission model. Inside Macintosh: Files describes how AppleShare does that on pages 2-17 and 2-18. If you want a more extensive permission model that AppleShare-aware applications can use, then you'll need to support the AppleShare HFSDispatch selectors and mimic AppleShare permissions. If you can't fit into the AppleShare model, you can come up with your own model and use PBGetForeignPrivs and PBSetForeignPrivs to manipulate your native privileges. However, you still must map those privileges to the HFS file system model. If you decide to use PBGetForeignPrivs and PBSetForeignPrivs, you'll use the creator type assigned to your foreign file system for your vMForeignPrivID number (the number you return for PBHGetVolParms requests).

Q: What is needed for a foreign file system to support PBCatSearch?

A: You'll need to support PBHGetVolParms so that you can indicate that you support CatSearch. It will be up to your foreign file system to come up with a fast way to search a volume’s catalog for matches. The Apple Developer Support sample code MoreFiles version 1.2 and later contains much of the code needed to implement CatSearch in the file Search.C.

Q: FindFolder seems to make some assumption on volumes that support AppleShare (AFP) access-control. What are those assuptions?

A: The folder manager assumes that user ID 0 is the guest user and user ID 1 is the administrator (owner) user and uses byte range locks on an empty data file as a semaphore to control access to individual trash folders on the volume. For more information, see Inside AppleTalk, second edition and the AppleTalk Filing Protocol Version 2.1 chapter of the AppleShare 3.0 Developer’s Kit.

Q: How should I handle VolumeMount requests?

A: VolumeMount requests are caught by the File System Manager before they are passed to the File Manager. FSM calls each installed foreign file system’s fileSystemCommProc with a ffsIDVolMountMessage. If the media type is not your foreign file system’s media type, you return extFSErr. If the media type is your foreign file system’s media type, you return noErr and the your HFSCIProc will be called with the VolumeMount select code.

When your HFSCIProc is called with the VolumeMount select code, your foreign file system should do everything needed to mount a volume. It should:

- Allocate memory for a volume control block (VCB) with UTAllocateVCB.
- Fill in the fields of the VCB.
- Allocate memory for a drive queue element (DrvQEl) in the system heap.
- Fill in the fields of the DrvQEl.
- Call _AddDrive to add the DrvQEl to the drive queue (DrvQHdr).
- Add the VCB to the VCB queue (VCBQHdr) with UTAddNewVCB.
- Use PostEvent to post a diskInsertEvt event.

The volume reference number you return to VolumeMount is the volume reference number returned by UTAddNewVCB.

After VolumeMount returns to the caller, the Event Manager will handle the diskInsertEvt event and will call your foreign file system with a _MountVol request. Since the volume is already mounted, your MountVol code only needs to return noErr.

Q: Can I put up an additional user authentication dialog when my foreign file system gets a VolumeMount request?

A: You put up an additional user authentication dialog if the caller of VolumeMount tells you it is safe to do that. To allow foreign file system to determine if it is safe, Apple has extended the VolMountInfoHeader record to include a flags word (the same flags word already in the AFPVolMountInfo record). By setting the volMountInteractBit (bit 15) in the flags word, the caller can tell your foreign file system that it is safe to use the Dialog Manager to put up your authentication dialog. In addition, if your foreign file system finds that the VolMountInfo record passed is usable but needs to be updated, it can set the volMountChangedBit (bit 14) in the flags word to tell the caller that it needs to update the VolMountInfo record using the PBGetVolMountInfoSize and PBGetVolMountInfo functions.

The Alias Manager has been updated to set the volMountInteractBit if the kARMNoUI rule is passed to the MatchAlias function, and to return the state of the volMountChangedBit in the needsUpdate parameter of the MatchAlias function and the wasChanged parameter the ResolveAlias function.

Here are the new constant definitions and data types you need:

```
/* The new volume mount flags */
enum {
   volMountInteractBit = 15,       /* Input to VolumeMount: If
                                      set, it's OK for the file
                                      system to perform user
                                      interaction to mount the
                                      volume */
   volMountInteractMask = 0x8000,

   volMountChangedBit = 14,        /* Output from VoumeMount: If
                                      set, the volume was mounted,
                                      but the volume mounting
                                      information record needs to
                                      be updated. */
   volMountChangedMask = 0x4000,

   volMountFSReservedMask = 0x00ff, /* bits 0-7 are defined each
                                       file system's use */
   volMountSysReservedMask = 0xff00 /* bits 8-15 are reserved for
                                       Apple system use */
};

/* The new volume mount info record */
struct VolMountInfoHeader {
   short         length;           /* length of location data
                                      (including self) */
   VolumeType    media;            /* type of media.  Variable
                                      length data follows */
   short         flags;            /* flags */
};
typedef struct VolMountInfoHeader VolMountInfoHeader;
typedef VolMountInfoHeader *VolMountInfoPtr;
```

Q: What is the correct time to put up an additional user authentication dialog when my foreign file system gets a VolumeMount request?

A: The correct time to put up an additional user authentication dialog is when your foreign file system's fileSystemCommProc gets the ffsIDVolMountMessage. You don't want to wait until your HFSCIProc is called because at that time, you're in the middle of a File Manager request and cannot do anything that might cause another File Manager request.

Q: How do I register a VolumeMount media type? How do I register a foreign privileges ID number?

A: Use your file system ID number for your foreign privileges ID number and use the creator type that you registered for your file system ID number for the VolumMount media type. If for some reason, you need another VolumeMount media type or foreign privileges ID number, you can simply register another file system ID number. It will be up to you to document your file system ID number, foreign privileges ID number, and VolumeMount media type if you want developers to know about them. That's keeping with Apple's policy of keeping creator types, card ID numbers, ADEV ID numbers, etc. confidential. If you decide to publish information specific to your file system, you should probably include header files for programmers and that's where you can document your numbers.

Q: Where are the constants for the new information I can get by calling Gestalt with the gestaltFSAttr selector?

A: Right here:

```
gestaltFSAttr = 'fs  ',            /* file system attributes */

gestaltFullExtFSDispatching = 0,   /* all HFSDispatch selectors
                                      are passed through to file
                                      systems */
gestaltHasFSSpecCalls = 1,         /* File Manager has FSSpec
                                      calls */
gestaltHasFileSystemManager = 2,   /* has the File System
                                      Manager */
gestaltFSMDoesDynamicLoad = 3,     /* File System Manager supports
                                      dynamic loading */
gestaltFSSupports4GBVols = 4,      /* file system supports
                                      4 gigabyte volumes */
gestaltFSSupports2TBVols = 5,      /* file system supports
                                      2 terabyte volumes */
gestaltHasExtendedDiskInit = 6,    /* has extended Disk
                                      Initialization calls */
```

Q: What should I return when called with the GetVolParms HFSDispatch select code?

A: Funny you should ask. Here's how GetVolParms looks from the file system side of things:

The GetVolParmsInfoBuffer Record from file systems implementing PBHGetVolParms:

Offset	Field	Meaning
0	vMVersion	Version number of the GetVolParmsInfoBuffer record you return. Version 1 indicates you can return the vMVersion through vMServerAdr fields. Version 2 indicates you can return the vMVersion through vMForeignPrivID fields. Note: You must return no more than ioReqCount bytes to PBHGetVolParms requests. Indicate the actual number of bytes returned in ioActCount.
2	vMAttrib	A 32-bit quantity that encodes information about the volume attributes.
6	vMLocalHand	If supplied, a handle to private data kept by the Macintosh system for shared volumes while they are mounted. When you mount the volume, you must allocate a handle to a 2-byte block of memory in the system heap which is returned in vMLocalHand. When you unmount the volume, dispose of the handle. Set the bLocalWList bit in the vMAttrib field to indicate that this field is valid. See the description of bLocalWList for when you should supply the handle.
10	vMServerAdr	For file server volumes, this field contains the AppleTalk internet address of the file server that manages the volume. An application can inspect this field to tell which volumes belong to a particular file server; the value of this field is 0 if the volume does not belong to a file server. If you support PBHCopyFile, you must initialize this field. The Finder uses this field to determine if a volume is a network volume (non-zero = network volume). If you have a network volume that doesn't have an AppleTalk address, you need to use something that isn't zero and isn't a valid AppleTalk address - I suggest that you use $ffffffff (-1); $ffffffff is an illegal AppleTalk address so it won't ever be used as a real AppleTalk address.
14	vMVolumeGrade	The relative speed rating of the volume. The scale used to determine these values is currently uncalibrated, so until further notice, return 0 in this field.

18	vMForeignPrivID	The access privilege model supported by the volume. Currently two values are defined for this field: 0 represents a standard HFS volume that might or might not support the AFP privilege model; fsUnixPriv represents a volume that supports the A/UX privilege model. I recommend that you use 0 and try to map your privilege model to the AFP privilege model for maximum compatibility with Macintosh applications.

The vMAttrib Field Bits from file systems implementing PBHGetVolParms:

Bit	**Name**	**Meaning**
31	bLimitFCBs	The volume supports a small finite number of open files. Set this bit to indicate that applications should attempt to limit the number of files they keep open on this volume. For example, the Finder limits the number of file control blocks used during copying to 8 instead of 16 if this bit is set.
30	bLocalWList	Set this bit to indicate that the vMLocalHand field of the GetVolParmsInfoBuffer record is valid and the Finder's open window list information is not stored on this volume. Shared volumes should set this bit.
29	bNoMiniFndr	Reserved; always set to 1.
28	bNoVNEdit	This volume's name cannot be edited. The System 7 Finder ignores this bit and looks at bAccessCntl instead for some reason.
27	bNoLclSync	Set this bit to indicate that modification dates on this volume can change with no action from this system. For example, a network volume's modification dates can change.
26	bTrshOffLine	Set this bit to indicate that this volume does not support an offline state. The PBOffLine function is not supported by this volume. If the Finder finds this volume offline, it is zoomed to the Trash and unmounted.
25	bNoSwitchTo	Obsolete. The Finder will not switch launch to any application on this volume.
24–21		Reserved. Set to zero.
20	bNoDeskItems	Set this bit to indicate that this volume does not support objects on the desktop.If the volume has a folder named Desktop Folder in it's root directory, it is treated as any other directory.
19	bNoBootBlks	Set this bit to indicate that this volume is not a startup volume. The Startup menu item is disabled. Boot blocks are not copied during copy operations.

18	bAccessCntl	Set this bit to indicate that this volume supports AppleTalk AFP access-control interfaces. The PBHGetLoginInfo, PBHGetDirAccess, PBHSetDirAccess, PBHMapID, and PBHMapName functions are supported. Special folder icons are used. The Access Privileges (System 6) or Sharing (System 7) menu item is enabled for disk and folder items. The ioACUser field returned by PBGetCatInfo for folders is assumed to be valid. Note: volumes with bAccessCntl and vMServerAdr=0 cannot be unmounted by the Finder and volumes with bAccessCntl cannot be renamed.
17	bNoSysDir	This volume doesn't support a system directory. Do not switch launch to this volume. The system directory location is not stored in ioVFndrInfo.
16	bHasExtFSVol	The bit name isn't accurate. What this bit says is that the Disk Initialization Manager is not supported by this volume's file system. The Finder disables Erase Disk item when this bit is set.
15	bHasOpenDeny	This volume supports the PBHOpenDeny and PBHOpenRFDeny functions.
14	bHasCopyFile	This volume supports the PBHCopyFile function, which is used in copy and duplicate operations if both source and destination volumes have the same server address in vMServerAdr.
13	bHasMoveRename	This volume supports the PBHMoveRename function.
12	bHasDesktopMgr	This volume supports the Desktop Manager functions (described in the chapter "Desktop Manager" in Inside Macintosh: More Macintosh Toolbox).
11	bHasShortName	This volume supports AFP short names with the GetXCatInfo select code ($3A). Once again, the bit name isn't really accurate.
10	bHasFolderLock	This volume supports PBHSetFLock and PBHRstFLock on folders. A locked folder cannot be deleted or renamed.
9	bHasPersonalAccessPrivileges	This volume has local file sharing enabled.
8	bHasUserGroupList	The user and group list of the local file server returned by the PBGetUGEntry function are valid for this volume.
7	bHasCatSearch	This volume supports the PBCatSearch function.
6	bHasFileIDs	This volume supports the file ID reference functions, including the PBExchangeFiles function.
5	bHasBtreeMgr	Reserved for internal Apple use.
4	bHasBlankAccessPrivileges	This volume supports inherited AFP access privileges for folders.
3–0		Reserved. Set them to zero.

Q: Can I write a native Power PC foreign file system with the File System Manager?

A: We do not recommend writing File System Manager-based foreign file systems in native Power PC code because there would be a minimum of two mixed mode switches per file system request you handle (you'll be called from emulated 68K code). Complex file system requests that cause multiple driver requests could cause many more mixed mode switches. If you think of a place where native code would really help the performance of your file system, you'd be best off putting just that code in a Power PC accelerated code resource and calling it from your foreign file system (which would still be mostly in 68K code) only when you need it.